***Spring*** – это свободно распространяемый фреймворк. Он способствует упрощению разработки на *java* c помощью:

* Легковесности и применению простых *Java*-объектов (*Plain-Old Java Objects,* *POJO*). Он никогда не заставляет обеспечивать реализацию своих интерфейсов или наследовать свои классы.
* Слабому связыванию посредством внедрения зависимостей и ориентированности на интерфейсы.
* Декларативному программированию через аспекты и общепринятые соглашения.
* Уменьшению объема типового кода через аспекты и шаблоны.

***Dependency Injection*** (**внедрение зависимостей**) позволяет внедрять зависимости в объекты от третьей стороны, вместо того чтобы объекты самостоятельно выстраивали связи между собой.

Внедрение зависимостей через конструктор:

*public class BraveKnight implements Knight {*

*private Quest quest;*

*public BraveKnight(Quest quest) { // Внедрение сценария подвига извне*

*this.quest = quest; }*

*public void embarkOnQuest() throws QuestException {*

*quest.embark(); } }*

**Связывание** (***wiring***) – это процесс создания связей между прикладными компонентами ***bean*** (например, между *BraveKnight* и *Quest*). Наиболее общим из них является способ на основе *XML*.

*<!-- Внедрение компонента quest -->*

*<bean id="knight" class="com.springinaction.knights.BraveKnight"> <constructor-arg ref="quest" /> </bean>*

*<!-- Создание SlayDragonQuest -->*

*<bean id="quest" class="com.springinaction.knights.SlayDragonQuest" />*

**Контекст приложения** загружает определения компонентов (объектов) и связывает их вместе.

*public class KnightMain {*

*public static void main(String[] args) {*

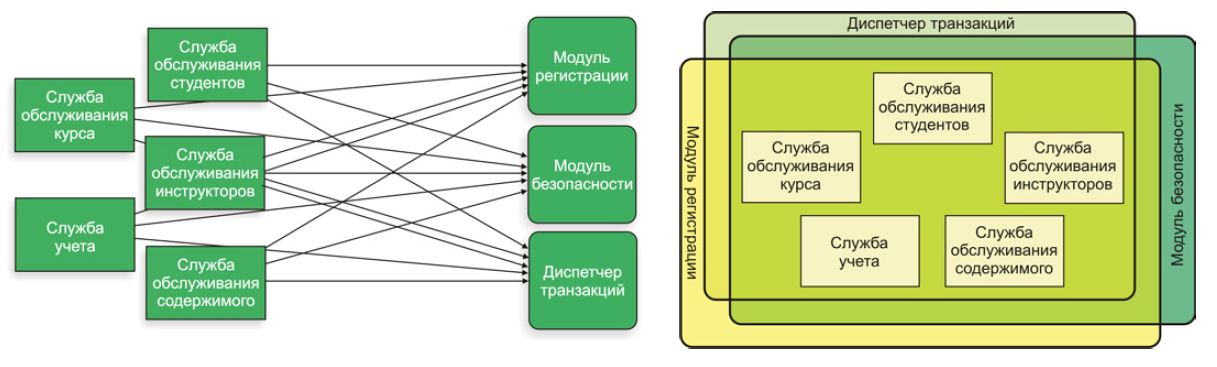
*ApplicationContext context = new ClassPathXmlApplicationContext("knights.xml"); // Загрузка контекста*

*Knight knight = (Knight) context.getBean("knight"); // Получение компонента knight*

*knight.embarkOnQuest(); } } // Использование компонента knight*

**Аспектно-ориентированное программирование** (***AOP***) – это прием, позволяющий оформить функциональность в виде многократно используемых компонентов.

Системные службы, которые используют компоненты дополнительно помимо их прямого назначения, называются **сквозными задачами**, потому что в их работу могут вовлекаться несколько компонентов системы. С помощью аспектов *AOP* позволяет отделить такие службы и их декларативное применение к компонентам. **Аспекты** можно представить как обертки, охватывающие множество компонентов в приложении, которые могут даже не подозревать об их существовании.



Создадим объект *Minestrel*:

*public class Minstrel {*

*public void singBeforeQuest() { // Вызывается перед выполнением задания*

*System.out.println(“The knight is so brave!”); }*

*public void singAfterQuest() { // Вызывается после выполнения задания*

*System.out.println(“The brave knight did embark on a quest!”); } }*

Объявим *Minestrel* компонентом и аспектом, заставив его отслеживать вызов метода класса *BraveKnight* с помощью XML разметки:

*<!-- Объявление компонента Minstrel -->*

*<bean id="minstrel" class="com.springinaction.knights.Minstrel" />*

*<aop:config>*

*<aop:aspect ref="minstrel">*

*<!-- Объявление точки внедрения -->*

*<aop:pointcut id="embark" expression="execution(\* \*.embarkOnQuest(..))" />*

*<!-- Операция, выполняемая до -->*

*<aop:before pointcut-ref="embark" method="singBeforeQuest"/>*

*<!-- Операция, выполняемая после -->*

*<aop:after pointcut-ref="embark" method="singAfterQuest"/>*

*</aop:aspect>*

*</aop:config>*

В результате компонент *Minestrel* остался простым *Java*-объектом и был применен к компоненту *BraveKnight* вообще без его ведома.

Для устранения шаблонного кода в *Spring* предусмотрены **классы-шаблоны**. Например, *JdbcTemplate* используется для упрощения работы с БД *jdbc*, автоматизируя создание подключения, определение и выполнение *SQL*-запроса, освобождение ресурсов и закрытие соединения.

Фреймворк *Spring* имеет две реализации контейнера:

* **Фабрики компонентов** (***bean factories***) – самые простые контейнеры, обеспечивающие поддержку DI. Они определяются интерфейсом *org.springframework.beans.factory.BeanFactory*.
* **Контекст приложений** (***application contexts***) – основан на понятии фабрик компонентов и реализует прикладные службы фреймворка (например, прием текстовых сообщений из файлов свойств). Они определяются интерфейсом *org.springframework.context.ApplicationContext*.

Далее будем рассматривать контексты приложений, т.к. фабрики компонентов слишком низкоуровневый инструмент.

Наиболее частые разновидности контекстов приложений (по местоположению *XML*-файла):

* *ClassPathXmlApplicationContext* – загружает контекст из библиотеке классов (*classpath*).
* *FileSystemXmlApplicationContext* - загружает контекст из в файловой системы.
* *XmlWebApplicationContext* – загружает контекст из содержимого веб-приложения.

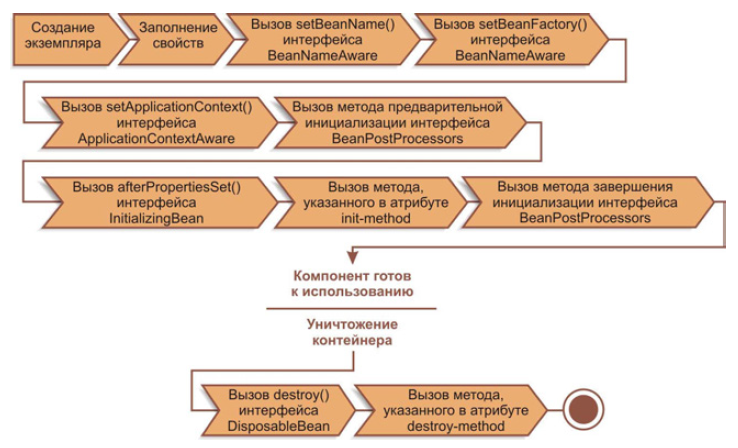
Пример загрузки контекста:

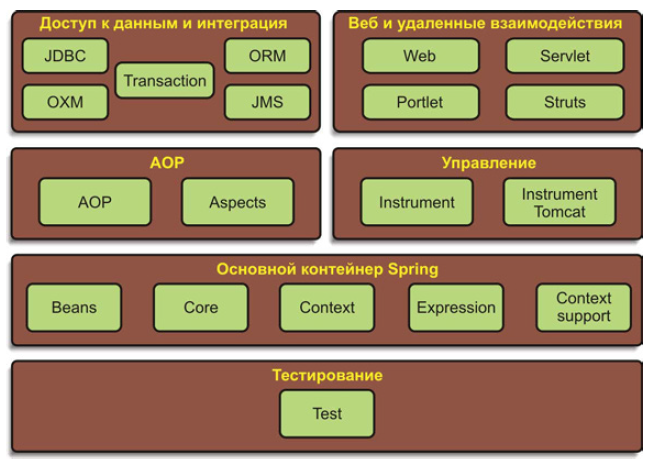
*ApplicationContext context = new FileSystemXmlApplicationContext("c:/foo.xml");*

*ApplicationContext context = new ClassPathXmlApplicationContext("foo.xml");*

После загрузки контекста приложения можно извлекать компоненты из контейнера *Spring* с помощью метода контекста *getBean()*.

Жизненный цикл экземпляра компонента внутри контейнера *Spring* сильно отличается от жизненного цикла обычного экземпляра объекта. Если компонент реализует указанные интерфейсы, то соответствующие им методы вызываются на этапе создания экземпляра компонента.





Фреймворк *Spring* состоит из нескольких модулей:

* **Основной контейнер** управляет процессом создания и настройки компонентов приложения. Он содержит фабрику компонентов, контекст приложения и несколько корпоративных служб, таких как электронная почта, доступ к *JNDI*, интеграция с *EJB* и выполнение заданий по расписанию.
* **Модуль AOP** служит основой при разработке аспектов в приложении, позволяет отделять аспекты приложения от объектов, к которым они применяются. Также способствует ослаблению связей между прикладными объектами.
* **Доступ к данным и интеграция** позволяет абстрагировать шаблонный код при операциях с БД и предотвратить проблемы при освобождении ресурсов. Также предоставляет рычаги управления другими популярными фреймворками и абстракцию интерфейса доступа к службам обмена сообщений (*Java Message Service*) для обеспечения асинхронных действий с другими приложениями. Начиная с версии *Spring* 3.0, включает еще и механизм отображение объектов в формат *XML*.
* **Веб и удаленные взаимодействия** позволяет использовать фреймворки *MVC* (*Model - View - Controller*). Также имеет собственные *MVC* фреймворки на основе сервлетов (для создания обычных приложений) и портлетов (для создания приложений на основе *API* *Java*-портлетов). Он также предоставляет поддержку удаленных взаимодействий для создания приложений, взаимодействующих с другими приложениями - механизм вызова удаленных методов (*Remote Method Invocation*) и собственный механизм вызова через *HTTP*.
* **Тестирование** содержит коллекцию фиктивных объектов для применения в модульных тестах, проверяющих работу с *JNDI*, сервлетами и портлетами.

Помимо основного фреймворка *Spring* существует множество дополнительные фреймворки и библиотеки, созданные на его основе:

* ***Spring Web Flow*** обеспечивает создание диалоговых многоэтапных веб-приложений, направляющих пользователя к его цели.
* ***Spring Web Services*** предлагает реализацию веб-служб по модели “*contract-first*”, когда программный код пишется на основе определения службы.
* ***Spring Security*** предлагает декларативный механизм обеспечения безопасности приложений.
* ***Spring Integration*** предлагает шаблоны интеграции приложений в декларативном стиле.
* ***Spring Batch*** – инструмент пакетной обработки данных.
* ***Spring Social*** обеспечивает интеграцию с сайтами социальных сетей.
* ***Spring Mobile*** обеспечивает поддержку мобильных приложений.
* ***Spring Dynamic Modules*** представляет собой сплав механизмов внедрения зависимостей *Spring* и динамических модулей *OSGi*.
* ***Spring LDAP*** предлагает шаблоны для механизма *LDAP*-операций.
* ***Spring Rich Client*** обеспечивает интеграцию *Spring* в *Swing* (создание настольных приложений).
* ***Spring.NET*** обеспечивает интеграцию *Spring* в *.NET*.
* ***Spring-Flex*** позволяет приложениям, созданным на основе *Flex* и *AIR*, взаимодействовать с серверными компонентами *Spring* посредством *BlazeDS*.
* ***Spring Roo*** реализует диалоговое окружение, обеспечивающее возможность быстрой разработки приложений на основе *Spring*.
* и другие расширения.

Начиная с версии 2.5 *Spring* стал поддерживать разработку на основе аннотаций, существенно уменьшающей объем *XML*-кода, необходимого для конфигурирования *Spring*. Начиная с версии 3.0 *Spring* стал полноценно поддерживать *REST*, углубил поддержку аннотаций и механизм отображения объектов (*Object-to-XML Mapping, OXM*).

Фреймворк *Spring* поддерживает несколько пространств имен XML, посредством которых выполняется настройка контейнера *Spring:*

|  |  |
| --- | --- |
| **Пространство имен** | **Назначение** |
| aop | Предоставляет элементы для объявления аспектов и для автоматического проксирования классов, объявляемых аспектами с помощью аннотации *@AspectJ* |
| beans | Пространство имен с основными элементами, позволяющими объявлять компоненты и определять связи между ними |
| context | Содержит элементы для конфигурирования контекста приложения *Spring*, включая возможность автоматического определения и автоматического связывания компонентов, а также внедрения объектов, которые не управляются фреймворком *Spring* непосредственно |
| jee | Обеспечивает интеграцию с такими *Java EE API*, как *JNDI* и *EJB* |
| jms | Предоставляет элементы для объявления *POJO*, управляемых сообщениями |
| lang | Позволяет объявлять компоненты, реализованные как сценарии на языках *Groovy*, *JRuby* и *BeanShell* |
| mvc | Включает такие возможности *Spring MVC*, как аннотированные контроллеры, контроллеры представлений и обработчики |
| oxm | Поддерживает возможность настройки механизма отображения объектов в *XML* |
| tx | Содержит элементы настройки декларативных транзакций |
| util | Набор различных вспомогательных элементов, включающий возможность объявления коллекций как компонентов и поддержку элементов-заполнителей свойств |

**Способы внедрения зависимостей:**

* Через конструктор
* Через *setter*
* Через конфигурации *scope*, *factory method* и другие
* Через *XML*, аннотации или *Java*-код
* Автоматически через *@Autowiring*

Рассмотрим пример создания компонентов через *XML*-файл конфигурации фреймворка *Spring*. По умолчанию он выглядит так:

*<?xml version="1.0" encoding="UTF-8"?>*

*<beans xmlns="http://www.springframework.org/schema/beans"*

*xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"*

*xsi:schemaLocation="http://www.springframework.org/schema/beans*

*http://www.springframework.org/schema/beans/spring-beans-3.0.xsd">*

*<!-- Здесь должны находиться объявления компонентов -->*

*</beans>*

Предположим, что все исполнители будут реализовывать соответствующий интерфейс:

*package com.springinaction.springidol;*

*public interface Performer {*

*void perform() throws PerformanceException; }*

Создадим класс исполнителя-жонглера, который реализует интерфейс исполнителя:

*package com.springinaction.springidol;*

*public class Juggler implements Performer {*

*private int beanBags = 3;*

*public Juggler() { }*

*public Juggler(int beanBags) {*

*this.beanBags = beanBags; }*

*public void perform() throws PerformanceException {*

*System.out.println("JUGGLING " + beanBags + " BEANBAGS"); } }*

В конфигурационном *XML*-файле *Spring* он будет объявлен следующим образом:

*<bean id="duke" class="com.springinaction.springidol.Juggler" />*

Здесь атрибут ***id*** присваивает компоненту имя, для дальнейшей работы с ним, а атрибут ***class*** сообщает фреймворку тип компонента. В данном случае используется конструктор по умолчанию (3 мяча).

Запустим представление через код в классе *main*:

*ApplicationContext ctx = new ClassPathXmlApplicationContext("com/springinaction/springidol/spring-idol.xml");*

*Performer performer = (Performer) ctx.getBean("duke");*

*performer.perform();*

В результате запуска такой программы получим строку “*JUGGLING 3 BEANBAGS*”.

Чтобы задействовать другой конструктор, изменим конфигурационный *XML*-файл *Spring*:

*<bean id="duke" class="com.springinaction.springidol.Juggler">*

*<constructor-arg value=”15”>*

*</bean>*

В результате запуска такой программы получим строку “*JUGGLING 15 BEANBAGS*”.

Создадим подкласс поэтического жонглера, который помимо жонглирования читает поэмы:

*public class PoeticJuggler extends Juggler {*

*private Poem poem; // Ссылка на итнерфейс поэмы*

*public PoeticJuggler(Poem poem) { // Внедрение поэмы*

*super(); // Вызов конструктора родительского класса*

*this.poem = poem; }*

*public PoeticJuggler(int beanBags, Poem poem) {*

*super(beanBags);*

*this.poem = poem; }*

*public void perform() throws PerformanceException {*

*super.perform();*

*System.out.println("While reciting...");*

*poem.recite(); } }*

Создадим интерфейс *Poem*:

*package com.springinaction.springidol;*

*public interface Poem {*

*void recite(); }*

Создадим класс *Sonnet29* реализующий интерфейс *Poem*:

*package com.springinaction.springidol;*

*public class Sonnet29 implements Poem {*

*private static String LINES = {"песня"};*

*public Sonnet29() {}*

*public void recite() {*

*System.out.println(LINES); } }*

Объявим объект класса *Sonnet29* компонентом *Spring* и изменим описание жонглера в *XML*-файле. Для передачи ссылок на другие компоненты используется атрибут ***ref***.

*<bean id="sonnet29" class="com.springinaction.springidol.Sonnet29" />*

*<bean id="poeticDuke" class="com.springinaction.springidol.PoeticJuggler">*

*<constructor-arg value="15" />*

*<constructor-arg ref="sonnet29" />* ***// Внедрение через конструктор***

*</bean>*

В результате запуска такой программы получим строку “*JUGGLING 15 BEANBAGS While reciting... песня*”.

Иногда единственный способ создать объект возможен только через **фабричный метод**.

Рассмотрим класс *Stage*, который реализован с применением шаблона проектирования “**Одиночка**” (***Singleton***), чтобы устранить любые возможности создания более одного его экземпляра. У него отсутствует конструктор, но есть метод, возвращающий один и тот же экземпляр класса.

*package com.springinaction.springidol;*

*public class Stage {*

***private*** *Stage() {}*

*private* ***static*** *class StageSingletonHolder {*

*static Stage instance = new Stage(); } // Создание экземпляра*

*public static Stage getInstance() {*

*return StageSingletonHolder.instance; } } // Возвращает экземпляр*

Чтобы превратить объект *Stage* в компонент *String*, достаточно воспользоваться атрибутом ***factory-method***:

*<bean id="theStage" class="com.springinaction.springidol.Stage" factory-method="getInstance" />*

При объявлении *<bean>* компонента можно определить его область действия с помощью атрибута ***scope***. Например:

*<bean id="ticket" class="com.springinaction.springidol.Ticket" scope="prototype" />*

Атрибут *scope* может получать следующие значения:

|  |  |
| --- | --- |
| **Область действия** | **Описание** |
| singleton | В каждом контейнере *Spring* может быть создан только один компонент (используется по умолчанию) |
| prototype | Позволяет создавать произвольное количество компонентов (по одному на каждое обращение) |
| request | Область действия компонента ограничивается *HTTP*-запросом. Может применяться только в веб-приложениях *Spring* (например, использующих *Spring MVC*) |
| session | Область действия компонента ограничивается *HTTP*-сеансом. Может применяться только в веб-приложениях *Spring* (например, использующих *Spring MVC*) |
| global-session | Область действия компонента ограничивается глобальным *HTTP*-сеансом. Может применяться только в портлетах |

Чтобы определить методы, вызываемые при создании и уничтожении компонента, нужно добавить в элемент *<bean>* атрибуты ***init-method*** и/или ***destroy-method***. Например, в концертном зале должны быть функционал включения/выключения света:

*public class Auditorium {*

*public void turnOnLights() { ... }*

*public void turnOffLights() { ... } }*

Добавим этот компонент в конфигурационный *XML*-файл *Spring*:

*<bean id="auditorium" class="com.springinaction.springidol.Auditorium"*

*init-method="turnOnLights" destroy-method="turnOffLights"/>*

В результате сразу после создания экземпляра компонента *Auditorium* будет вызван метод *turnOnLights()*, включающий освещение зала. А перед удалением компонента из контейнера будет вызван метод *turnOffLights()*, выключающий освещение.

Альтернативой такого способа является реализация интерфейсов ***InitializingBean*** и ***Disposablebean*** в классе компонента. Это позволит уменьшить количество *XML*-настроек, но создаст тесную связь.

Для множественного определения методов, вызываемых при создании и уничтожении объекта, можно использовать атрибуты ***default-init-method*** и ***default-destroy-method*** в элементе *<beans>*:

*<?xml version="1.0" encoding="UTF-8"?>*

*<beans xmlns="http://www.springframework.org/schema/beans"*

*xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"*

*xsi:schemaLocation="http://www.springframework.org/schema/beans*

*http://www.springframework.org/schema/beans/spring-beans-3.0.xsd"*

*default-init-method="turnOnLights"*

*default-destroy-method="turnOffLights"> …*

*</beans>*

Создадим класс музыканта, который реализует интерфейс исполнителя. Пусть свойство *song* хранит название мелодии, а свойство instrument хранит ссылку на экземпляр класса *Instrument*:

*package com.springinaction.springidol;*

*public class Instrumentalist implements Performer {*

*public Instrumentalist() {}*

*public void perform() throws PerformanceException {*

*System.out.print("Playing " + song + " : ");*

*instrument.play(); }*

*private String song;*

*public void setSong(String song) { // Внедрение мелодии*

*this.song = song; }*

*public String getSong() {*

*return song; }*

*public String screamSong() {*

*return song; }*

*private Instrument instrument;*

*public void setInstrument(Instrument instrument) { // Внедрение инструмента*

*this.instrument = instrument; } }*

Зададим класс *Instrument* с помощью интерфейса и создадим для него инструмент:

*package com.springinaction.springidol;*

*public interface Instrument {*

*public void play(); }*

*public class Saxophone implements Instrument {*

*public Saxophone() {}*

*public void play() {*

*System.out.println("TOOT TOOT TOOT"); } }*

Создадим компонент *Saxophone*:

*<bean id="saxophone" class="com.springinaction.springidol.Saxophone" />*

Для настройки свойств компонентов используется элемент ***property***. Он вызывает метод записи свойства ***set***. В атрибуте ***value*** можно указывать строковые, числовые и логические значения. Атрибут ***ref*** передает ссылку на компонент. Объявим музыканта как компонент *<bean>*:

*<bean id="kenny" class="com.springinaction.springidol.Instrumentalist">*

*<property name="song" value="Jingle Bells" />* ***// Внедрение через сеттеры***

*<property name="instrument" ref="saxophone" />*

*</bean>*

Запустим представление через код в классе *main*:

*ApplicationContext ctx = new ClassPathXmlApplicationContext("com/springinaction/springidol/spring-idol.xml");*

*Performer performer = (Performer) ctx.getBean("kenny");*

*performer.perform();*

В результате запуска такой программы получим строку “*Playing Jingle Bells : TOOT TOOT TOOT*”.

В итоге один и тот же код в методе *main()*, с помощью преимущества использования интерфейсов, заставляет выступать жонглера и музыканта. С помощью объявления компонентов в *XML*-файле можно легко изменить инструмент и мелодию, не изменяя код в используемых компонентах.

Для создания персонального музыкального инструмента конкретного исполнителя можно использовать **внутренние компоненты** (аналог вложенных классов для компонентов). Из недостатков – они не могут быть использованы повторно. Рассмотрим пример внедрения через элемент *property*:

*<bean id="kenny" class="com.springinaction.springidol.Instrumentalist">*

*<property name="song" value="Jingle Bells" />*

*<property name="instrument">*

*<bean class="org.springinaction.springidol.Saxophone" />*

*</property>*

*</bean>*

Внутренние компоненты можно внедрять также через аргументы конструктора:

*<bean id="duke" class="com.springinaction.springidol.PoeticJuggler">*

*<constructor-arg value="15" />*

*<constructor-arg>*

*<bean class="com.springinaction.springidol.Sonnet29" />*

*</constructor-arg>*

*</bean>*

**Для внедрения контекста из файла**, в файле конфигурации нужно добавить строку:

*<context:property-placeholder location=”classpath:musicPlayer.properties” />*

**Для связывания через аннотации**, в файле конфигурации нужно добавить строку:

*<context:component-scan base-package=”ru.alishev.sprngcourse” />*

*@Component* - помечает класс для создания из него бина. В скобках можно указать персональный id для создания бина (если его не указать, то его название будет *название\_класса\_с\_маленькой\_буквы*).

*@Autowired* – запускает **автоматическое связывание**. Позволяет внедрить зависимость в приватное поле через рефлексию (даже если нет конструктора или сеттера).

*@Qualifier* – позволяет уточнить какой именно бин нужно автоматически внедрить.

*@Scope* – позволяет задать область видимости бинов

*@Value* – позволяет внедрять контекст из файла. *@Value(“${musicPlayer.name}”)*

*@PostConstruct* – позволяет внедрить атрибут *init-method*. Может иметь любой модификатор доступа и не должен принимать аргументы.

*@PreDestroy* – позволяет внедрить атрибут *destroy-method*. Может иметь любой модификатор доступа и не должен принимать аргументы.

**Связывание только через *java*-код:**

В классе main нужно вызвать контекст следующей строкой:

*AnnotationConfigApplicationContext context = new AnnotationConfigApplicationContext(SpringConfig.class);*

*@Configuration* – позволяет сделать указанный класс *SpringConfig* конфигурационным.

*@ComponentScan* – позволяет искать классы помеченные *@Component* для создания бинов. Например, *@ComponentScan(”ru.alishev.sprngcourse”).*

*@Bean* – позволяет вручную создать бин.

*@PropertySource* – позволяет внедрять контекст из файла. Например, *@PropertySource(“classpath:musicPlayer.properties”).*

Для сокращения количества кода в *XML*-файле можно использовать **пространство имен *p***. Оно позволяет использовать для связывания свойств атрибуты с префиксом *p:* элемента *<bean>*. Например:

*<?xml version="1.0" encoding="UTF-8"?>*

*<beans xmlns="http://www.springframework.org/schema/beans"*

*xmlns:p="http://www.springframework.org/schema/p"*

*xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"*

*xsi:schemaLocation="http://www.springframework.org/schema/beans*

*http://www.springframework.org/schema/beans/spring-beans-3.0.xsd">*

*<bean id="kenny" class="com.springinaction.springidol.Instrumentalist"*

*p:song = "Jingle Bells"*

*p:instrument-ref = "saxophone" />*

Атрибуты *value* и *ref* можно использовать только когда свойства компонента содержат единственное значение. Элементы ***<list>*** и ***<set>*** можно использовать для настройки свойств, которые являются массивами или коллекциями. Элементы ***<map>*** и ***<props>*** соответствуют коллекциям с интерфейсами *java.util.Map* и *java.util.Properties*.

|  |  |
| --- | --- |
| **Элемент-коллекция** | **Назначение** |
| <list> | Связывание списка значений, допускаются повторяющиеся значения |
| <set> | Связывание множества значений, гарантирует отсутствие повторяющихся значений |
| <map> | Связывание коллекций пар имя/значение, где имя и значение могут быть значениями любых типов |
| <props> | Связывание коллекций пар имя/значение, где имя и значение должны имеет строковый тип (*String*) |

Создадим класс человека-оркестра, который реализует интерфейс исполнителя и играет сразу на нескольких инструментах:

*package com.springinaction.springidol;*

*import java.util.Collection;*

*public class OneManBand implements Performer {*

*public OneManBand() {}*

*public void perform() throws PerformanceException {*

*for (Instrument instrument : instruments) {*

*instrument.play(); } }*

*private Collection<Instrument> instruments;*

*public void setInstruments(Collection<Instrument> instruments) { // Внедрение коллекции инструментов*

*this.instruments = instruments; } }*

Создадим объект человека-оркестра и передадим ему набор инструментов. В данном случае элементы *<list>* и *<set>* являются взаимозаменяемыми:

*<bean id="hank" class="com.springinaction.springidol.OneManBand">*

*<property name="instruments">*

*<list>*

*<ref bean="guitar" />*

*<ref bean="cymbal" />*

*<ref bean="harmonica" />*

*</list>*

*</property>*

*</bean>*

Если бы вместо *Collection* мы использовали *Map*, то *XML*-файл выглядел иначе:

*<bean id="hank" class="com.springinaction.springidol.OneManBand">*

*<property name="instruments">*

*<map>*

*<entry key="GUITAR" value-ref="guitar" />*

*<entry key="CYMBAL" value-ref="cymbal" />*

*<entry key="HARMONICA" value-ref="harmonica" />*

*</map>*

*</property>*

*</bean>*

В зависимости от контекста можно заменять атрибуты *key* на *key-ref* и *value* на *value-ref*.

Если бы вместо коллекции строк и ссылок на компоненты в свойство объекта *OneManBand* внедряется отображение строки в строку *java.util.Properties*, то *XML*-файл выглядел так:

*<bean id="hank" class="com.springinaction.springidol.OneManBand">*

*<property name="instruments">*

*<props>*

*<prop key="GUITAR">STRUM STRUM STRUM</prop>*

*<prop key="CYMBAL">CRASH CRASH CRASH</prop>*

*<prop key="HARMONICA">HUM HUM HUM</prop>*

*</props>*

*</property>*

*</bean>*

Элемент *<prop>* задает ключ и его значение (инструмент и издаваемый им звук).

Чтобы присвоить свойству значение *null*, достаточно воспользоваться элементом *<null/>*:

*<property name="someNonNullProperty"><null/></property>*

**Язык выражений *Spring* (*Spring Expression Language, SpEL*)** – краткий и мощный способ внедрения значений в свойства компонентов или аргументы конструкторов с помощью выражений, которые вычисляются на этапе выполнения. Напомним, что в *XML*-файлах конфигурация задавалась на этапе разработки.

Для внедрения литералов в свойство компонентов в атрибуте value используется конструкция ***#{ }*** :

*<property name="message" value="The value is #{5}"/>*

*<property name="frequency" value="#{89.7}"/>*

*<property name="name" value="#{'Chuck'}"/>*

*<property name="enabled" value="#{false}"/>*

Этот конструктор также можно использовать для внедрения ссылок на другие компоненты. Например, следующий строки идентичны:

*<property name="instrument" ref="saxophone"/>*

*<property name="instrument" value="#{saxophone}"/>*

Создадим новый компонент исполнителя-подражателя на основе существующих классов:

*<bean id="carl" class="com.springinaction.springidol.Instrumentalist">*

*<property name="song" value="#{kenny.song}" />*

*</bean>*

Кроме возможности обращения к свойствам, можно вызывать методы компонентов. В данном примере используется защищенный оператор доступа к свойству ***?.*** , который защищает от *null*:

*<property name="song" value="#{songSelector.selectSong()?.toUpperCase()}"/>*

Для обращения к статическим свойствам и методам нужно использовать класс с оператором ***T( )*** . В результате вернется объект данного класса. Например:

*<property name="multiplier" value="#{T(java.lang.Math).PI}"/>*

*<property name="randomNumber" value="#{T(java.lang.Math).random()}"/>*

Язык выражений *Spring* включает следующие операторы для работы со значениями.

|  |  |
| --- | --- |
| **Типы операторов** | **Операторы** |
| Арифметические | +, -, \*, /, %, ^ (возведение в степень) |
| Операторы отношений | <, >, ==, <=, >=, lt, gt,eq,le,ge (текстовые лучше) |
| Логические | And, or, not, ! |
| Условные | ?: (тернарный), ?: (Элвис) |
| Регулярные выражения | matches |

Оператор + перегружен и может использоваться для конкатенации строк.

Оператор Элвиса работает аналогично нижней строке.

*<property name="song" value="#{kenny.song ?: 'Greensleeves'}"/>*

*<property name="song" value="#{kenny.song != null ? kenny.song : 'Greensleeves'}"/>*

Для сравнения строк используется оператор ***matches***.

Язык выражений *Spring* позволяет работать с коллекциями. Предположим есть класс:

*package com.habuma.spel.cities;*

*public class City {*

*private String name;*

*private String state;*

*private int population; }*

И конфигурационный файл со списком объектов *City*:

*<util:list id="cities">*

*<bean class="com.habuma.spel.cities.City" p:name="Chicago" p:state="IL" p:population="2853114"/>*

*<bean class="com.habuma.spel.cities.City" p:name="Atlanta" p:state="GA" p:population="537958"/>*

*<bean class="com.habuma.spel.cities.City" p:name="Dallas" p:state="TX" p:population="1279910"/>*

*</util:list>*

Извлечь из списка единственный элемент и внедрить его в свойство можно несколькими способами:

*<property name="chosenCity" value="#{cities[2]}"/> // как третий элемент ArrayList*

*<property name="chosenCity" value="#{cities['Dallas']}"/> // как ключ элемента Map*

*<util:properties id="settings" location="classpath:settings.properties"/> // как элемент Props*

*<property name="accessToken" value="#{settings['twitter.accessToken']}"/>*

Фреймворк *Spring* также позволяет получить доступ в *SpEL* к свойствам *systemEnvironment* и *systemProperties*. Первая содержит все переменные окружения системы, а вторая – все параметры запуска приложения.

Извлечь из списка множество элементов, соответствующих условию, и внедрить его в свойство можно с помощью операторов выборки ***.?[ ]*** , ***.^[ ]*** , ***.$[ ]*** :

*<property name="bigCities" value="#{cities.?[population gt 100000]}"/> // получить все элементы выборки*

*<property name="aBigCity" value="#{cities.^[population gt 100000]}"/> // получить первый элемент выборки*

*<property name="aBigCity" value="#{cities.$[population gt 100000]}"/> // получить последний элемент выб.*

Отображение коллекции связано с выбором определенного свойства каждого элемента и помещением его в новую коллекцию с помощью оператора отображения ***.![ ]*** . Например, создадим список строк с именами городов на основе списка объектов *City*:

*<property name="cityNames" value="#{cities.![name]}"/>*

Операторы выборки и отображения можно комбинировать:

*<property name="cityNames" value="#{cities.?[population gt 100000].![name + ', ' + state]}"/>*

Компоненты *<bean>* поддерживают **наследование**. Для этого могут использоваться атрибуты ***parent*** и **abstract**. Это помогает уменьшить объем *XML*-кода. Например, создадим абстрактный базовый класс и несколько его наследников:

*<bean id="baseSaxophonist"*

*class="com.springinaction.springidol.Instrumentalist"*

*abstract="true">*

*<property name="instrument" ref="saxophone" />*

*<property name="song" value="Jingle Bells" />*

*</bean>*

*<bean id="kenny" parent="baseSaxophonist" />*

*<bean id="frank" parent="baseSaxophonist">*

*<property name="song" value="Mary had a little lamb" /> // переопределим наследуемое свойство*

*</bean>*

В *Spring* подкомпонент **не обязан наследовать тип родителя**. Два компонента с разными значениями атрибута *class* могут иметь общие настройки свойств, унаследованные от родительского компонента. Это возможно, если в родительском компоненте не определен атрибут *class*. Например:

*<bean id="basePerformer" abstract="true">*

*<property name="song" value="Somewhere Over the Rainbow" />*

*</bean>*

*<bean id="taylor" class="com.springinaction.springidol.Vocalist" parent="basePerformer" />*

*<bean id="stevie" class="com.springinaction.springidol.Instrumentalist" parent="basePerformer">*

*<property name="instrument" ref="guitar" />*

*</bean>*

Кроме внедрения в методы значений свойств, *Spring* поддерживает **внедрение методов**. Внедрение методов бывает двух типов - **замещение метода** и **внедрение метода чтения**.

Создадим класс фокусника, который изменяет объект в черном ящике:

*package com.springinaction.springidol;*

*public class Magician implements Performer {*

*public Magician() {}*

*public void perform() throws PerformanceException {*

*System.out.println(magicWords);*

*System.out.println("The magic box contains...");*

*System.out.println(magicBox.getContents()); } // Исследует содержимое ящика*

*private MagicBox magicBox;*

*public void setMagicBox(MagicBox magicBox) {*

*this.magicBox = magicBox; } // Внедрение волшебного ящика*

*private String magicWords;*

*public void setMagicWords(String magicWords) {*

*this.magicWords = magicWords; } }*

Зададим реализацию черного ящика:

*public class MagicBoxImpl implements MagicBox {*

*public MagicBoxImpl() {}*

*public String getContents() {*

*return "A beautiful assistant"; } } // Симпатичная ассистентка в ящике*

Настроим *XML*-файл конфигурации:

*<bean id="harry" class="com.springinaction.springidol.Magician">*

*<property name="magicBox" ref="magicBox" />*

*<property name="magicWords" value="Bippity boppity boo" />*

*</bean>*

*<bean id="magicBox" class="com.springinaction.springidol.MagicBoxImpl" />*

В результате получим следующий вывод:

*Bippity boppity boo*

*The magic box contains...*

*A beautiful assistant*

Зададим подменный метод с тигром:

*package com.springinaction.springidol;*

*import java.lang.reflect.Method;*

*import org.springframework.beans.factory.support.MethodReplacer;*

*public class TigerReplacer implements MethodReplacer {*

*public Object reimplement(Object target, Method method, Object[] args) throws Throwable {*

*return "A ferocious tiger"; } } // Помещает тигра в ящик*

Изменим настройку *XML*-файла:

*<bean id="magicBox" class="com.springinaction.springidol.MagicBoxImpl">*

*<replaced-method name="getContents" replacer="tigerReplacer" />*

*</bean>*

*<bean id="tigerReplacer" class="com.springinaction.springidol.TigerReplacer" />*

Элемент *<replaced-method>* осуществляет **замещение метода**, где *name* – имя старого метода, а *replacer* – ссылка на компонент нового метода.

В результате получим следующий вывод:

*Bippity boppity boo*

*The magic box contains...*

*A ferocious tiger*

Продолжение стр.131+169+199+238+297+322+393+433+475+510+556+589+612+661+683+718

@Controller – унаследован от @Component и имеет доп. возможности. Помечает класс как контроллер.

@GetMapping, @PostMapping, @PutMapping, @DeleteMapping, @PatchMapping – связывает метод контроллера с адресом, по которому можно к этому методу обратиться. Вид маппинга зависит от вида запроса.

@RequestMapping(“/people”) – при использовании на классе, добавляет в URL адрес всех методов префикс /people.

<a href=”/goodbye”>Say goodbye</a> - текст со ссылкой на другой метод /goodbye контроллера

Статусы ответа:

200 – все ОК

3xx – нужен редирект

4xx – ошибка на стороне клиента

5xx – ошибка на стороне сервера

**GET-запрос** – это запрос на получение данных с сервера без изменения данных на сервере (идемпотентный запрос). Его тело всегда пустое, а все параметры передаются в URL после знака ? в формате ключ=значение. Несколько параметров разделяются знаком &. Например:

https://vk.com/audios33064131?***q=Gershwin***&***section=playlists***

Обработать запрос с параметрами можно либо сервлетом, либо через аннотацию @RequestParam :

@GetMapping(“/hello”)

public String helloPage(HttpServletRequest request) {

String name = request.getParameter(“name”);

return “hello”; }

@GetMapping(“/hello”)

public String helloPage(@RequestParam(“name”) String name) {

return “hello”; }

В случае с аннотацией, при отсутствии параметра страница выпадет с ошибкой. Чтобы избежать этого можно использовать следующий синтаксис присваивания *null* при отсутствии параметров:

@RequestParam(value = “name”, required = false) String name

**POST-запрос** – это запрос на отправку данных на сервер для изменения данных. Его URL всегда пустое, а все параметры передаются в теле запроса в любом формате.

Автоматическое внедрение объекта модели в метод контроллера:

@GetMapping(“/hello”)

public String helloPage(Model model) {

model.addAttribute(“message”, “Hello!”);

return “hello”; }

Обратиться в представлении к значению по ключу:

<p th:text=”${message}”></p>

**CRUD** (Create/Read/Update/Delete) – стандартная классификация функций по манипуляции данными. **CRUD приложение** – это приложение, реализующее весь этот функционал. Рассмотрим общепринятую конвенцию именования запросов URL на примере сущности *Post*:



**REST** – это паттерн проектирования web-приложений. Он описывает, как по протоколу HTTP взаимодействовать с сервером для работы с данными (реализует концепцию **CRUD**). Плюс много чего другого.

**DAO** (**Data Access Object**) – это паттерн проектирования, когда логика взаимодействия с БД вынесена в отдельный класс.

Обратиться к динамическому пути можно с помощью аннотации @PathVariable:

@GetMapping(“/{id}”)

public String show(@PathVariable(“id”) int id) {

return “show”; }

@Autowired – автоматическое внедрение компонента.

Вывод списка в представлении + динамические ссылки на каждого человека:

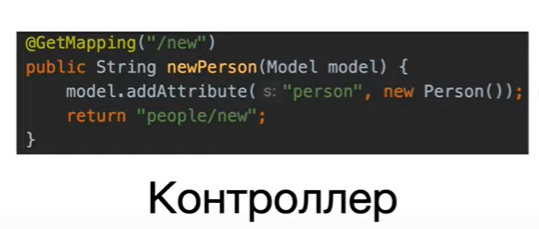
<div th:each=”person : ${people}”>

<a th:href=”@{/people/{id}(id=${person.getId()})}” th:text=”${person.getName()}”></a>

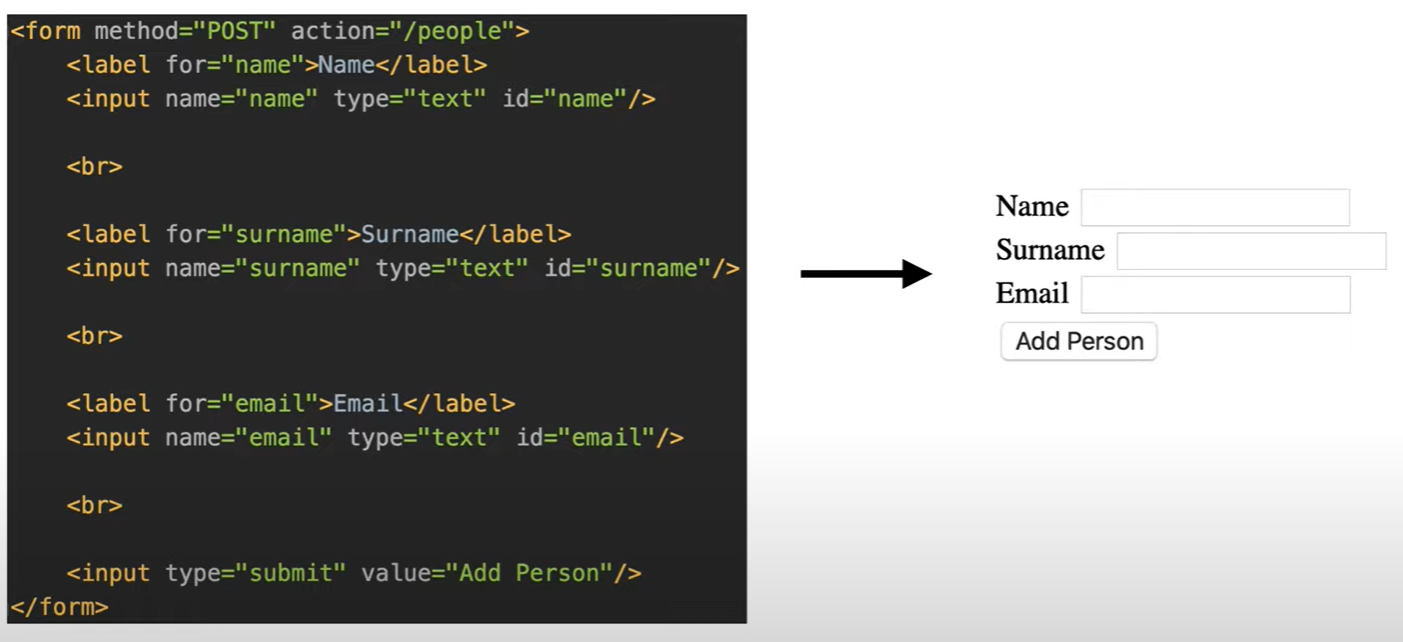
</div>

Создание формы POST-запроса:

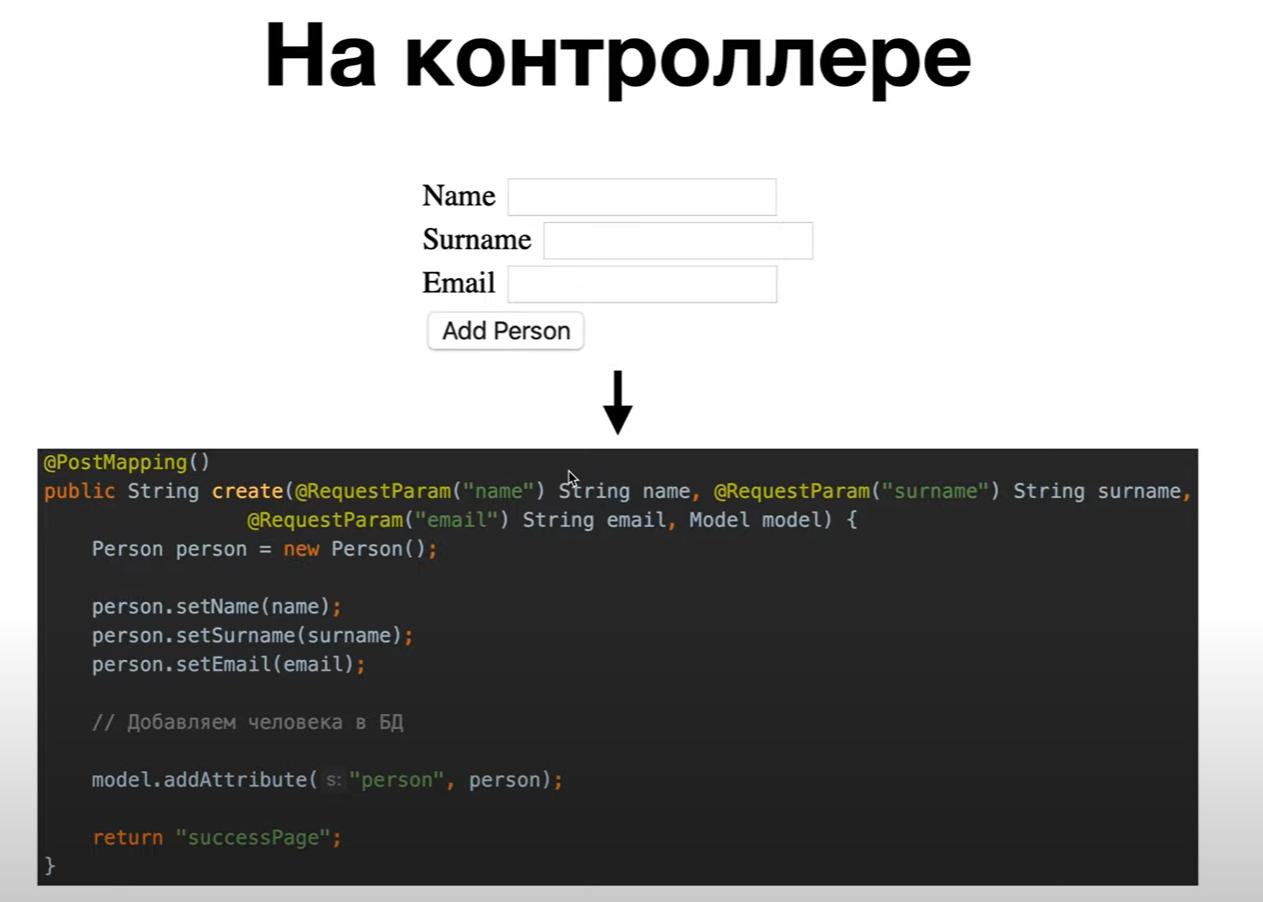
1. Создать контроллер перенаправляющий на форму



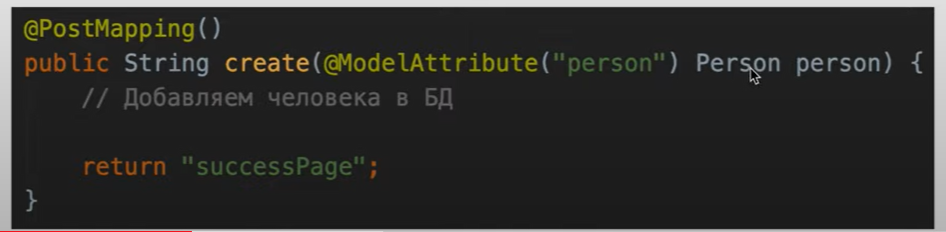
1. Создать представление, которое рисует форму.



1. Создать метод, вызывающийся при отправке формы:



@ModelAttribute – при использовании в аргументе метода, автоматизирует внедрение полученных данных в модель. Например, перепишем контроллер формы person выше:



Если не передать поля класса Person через форму, эти поля будут заполнены полями по умолчанию.

@ModelAttribute – при использовании на методе, внедряет во все создаваемые объекты модели данного контроллера пару ключ-значение. Например, внедрим “headerMessage”=”Welcome”:

@ModelAttribute(“headerMessage”)

public String populateHeaderMessage() {

return “Welcome”; }

Обращение к полю объекта, внедренного в модель, и присваивание ему id для сопоставления с надписью label:

<form th:method=”PATCH” th:action=”@{/people/{id}(id=${person.getID()})” th:object=”$person}”>

<label for=”name”>Enter name: </label>

<input th:”text” th:field=”\*{name}” id=”name”>

</form>

return “redirect:/people” - перенаправление на другую страницу.

HTML 5 поддерживает только методы GET и POST. Для использования других методов, которые поддерживает HTTP, на стороне представления используется скрытое поле:

<form method=”post” action=”/people/1”>

<input type=”hidden” name=”\_method” value=”patch”>

</form>

А на стороне контроллера используются фильтры, подключаемые в конфигурационных файлах.

Для включения валидации форм нужно внедрить зависимость Hibernate Validator и использовать аннотации на полях:

@NotEmpty(message = “Поле не может быть пустым”)

@Size(min = 2, max = 30, message = “Введите от 2 до 30 символов”)

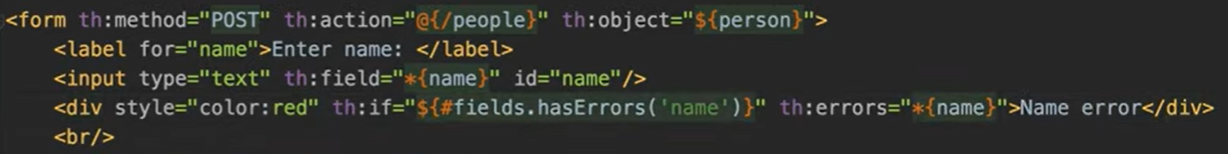
@Min(value = 0, message = “Число должно быть неотрицательным”)

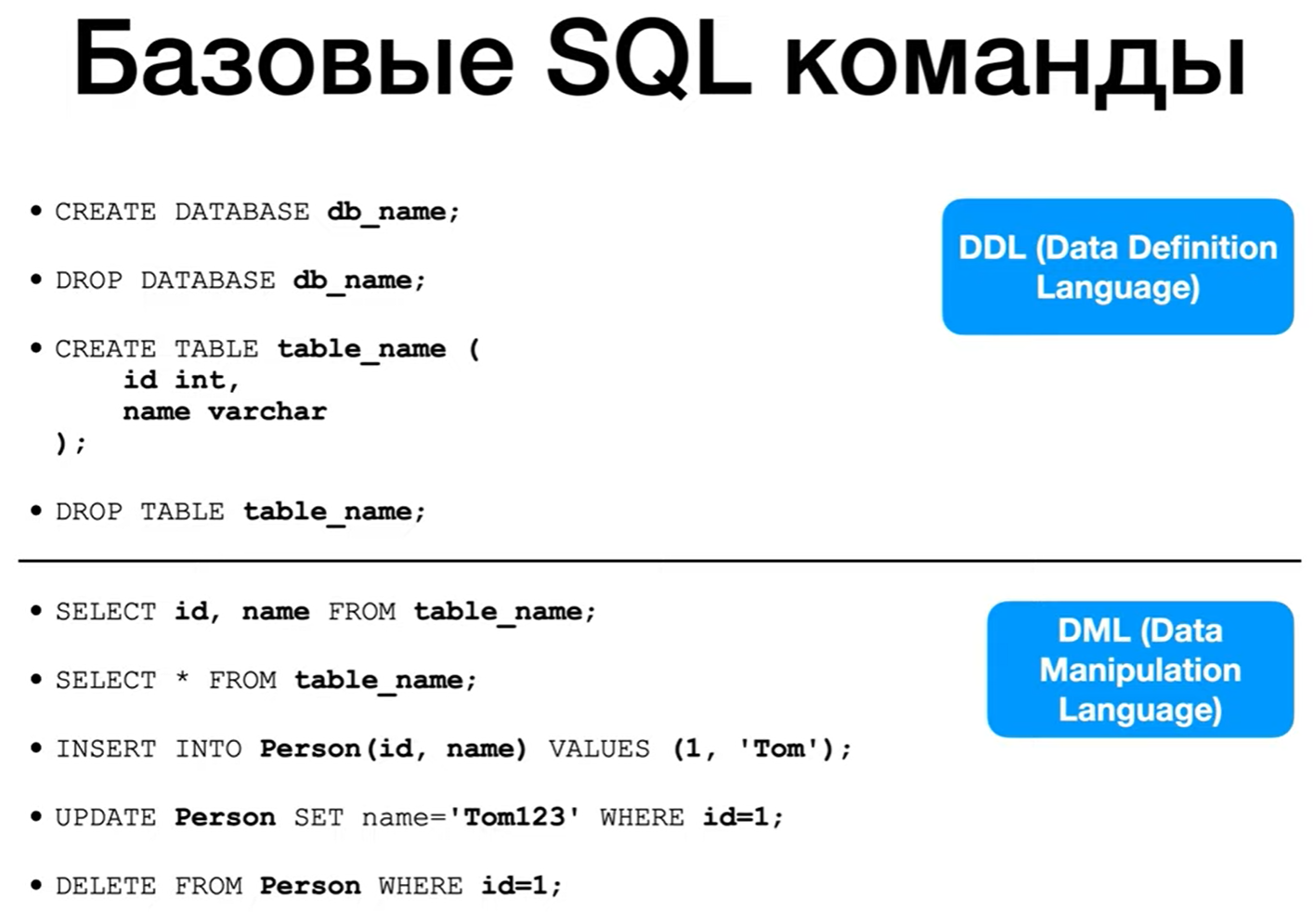
@Email(message = “Не валидная почта“)

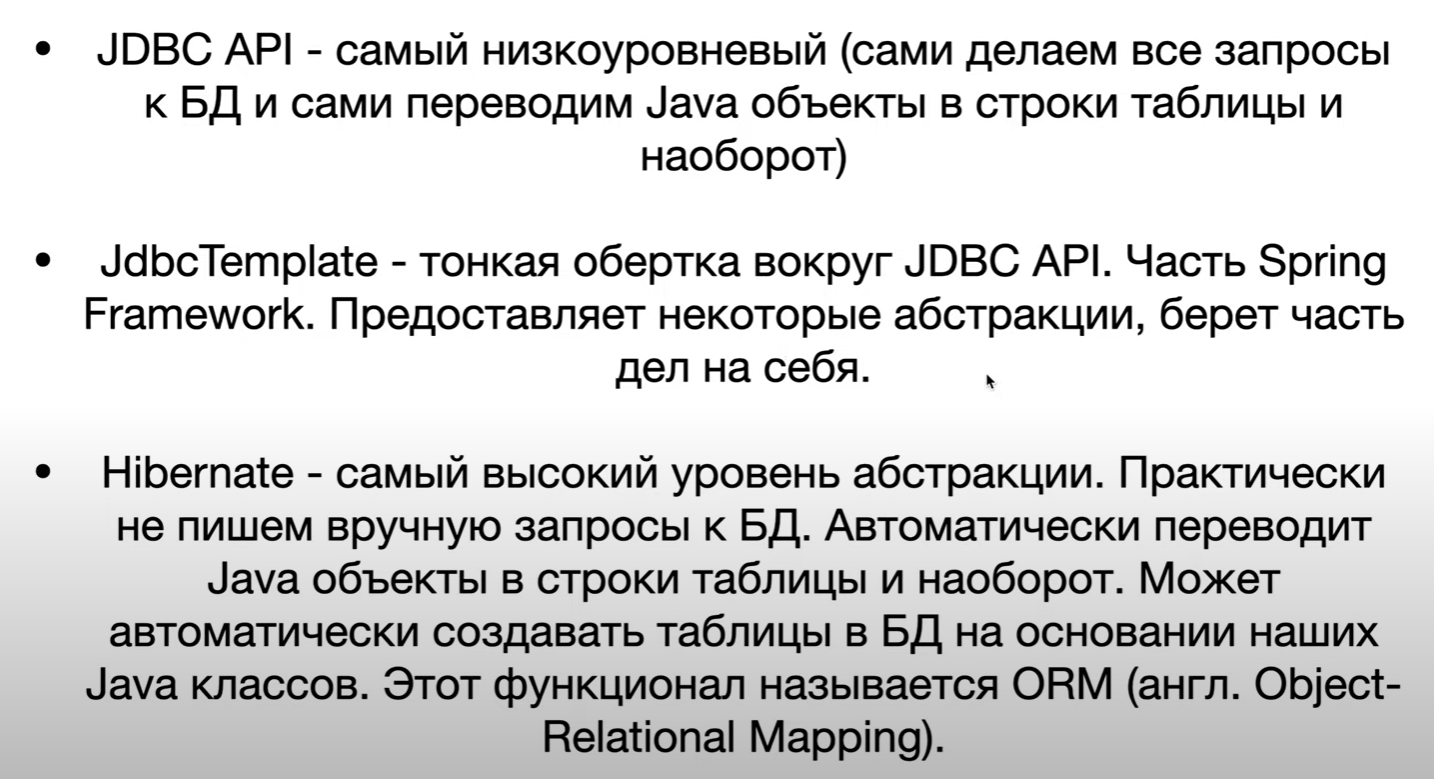
А в методе используем аннотацию @Valid для проверки объекта и его полей, ошибка валидации будет сохраняться в объект BindingResult:



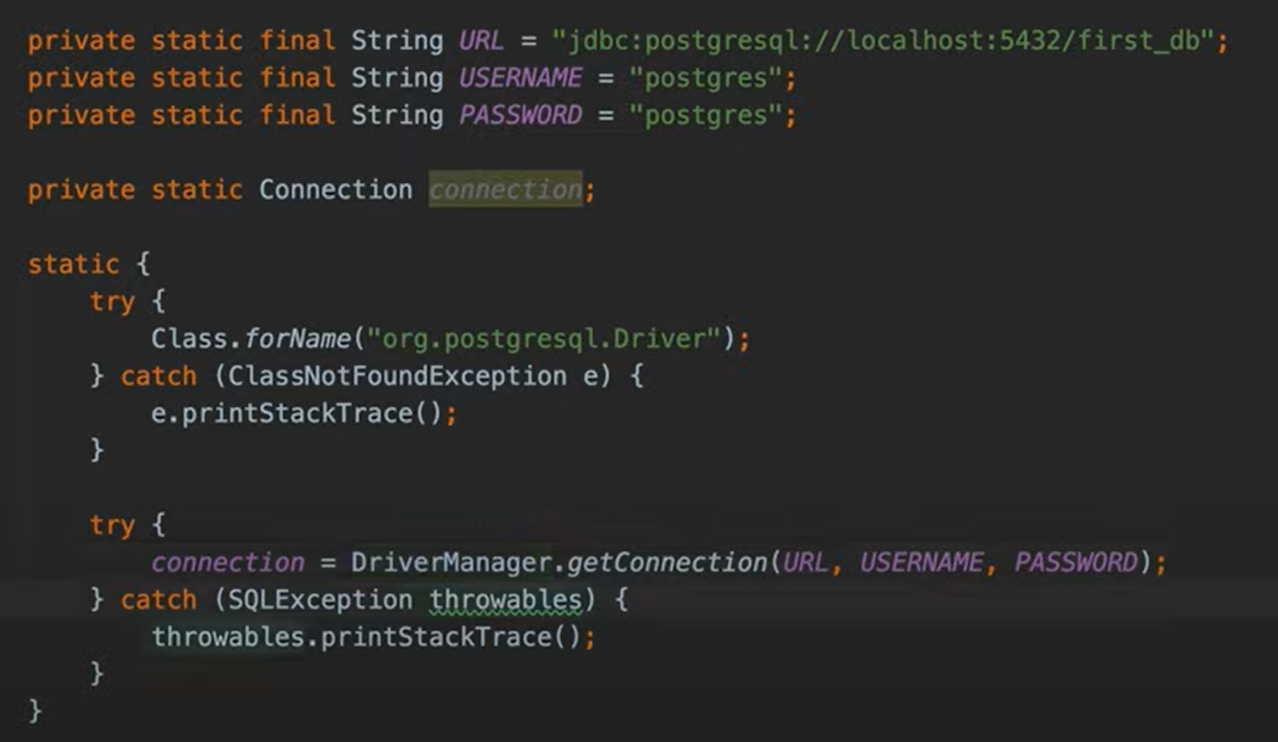
Динамически выведем ошибки валидации для каждого поля:



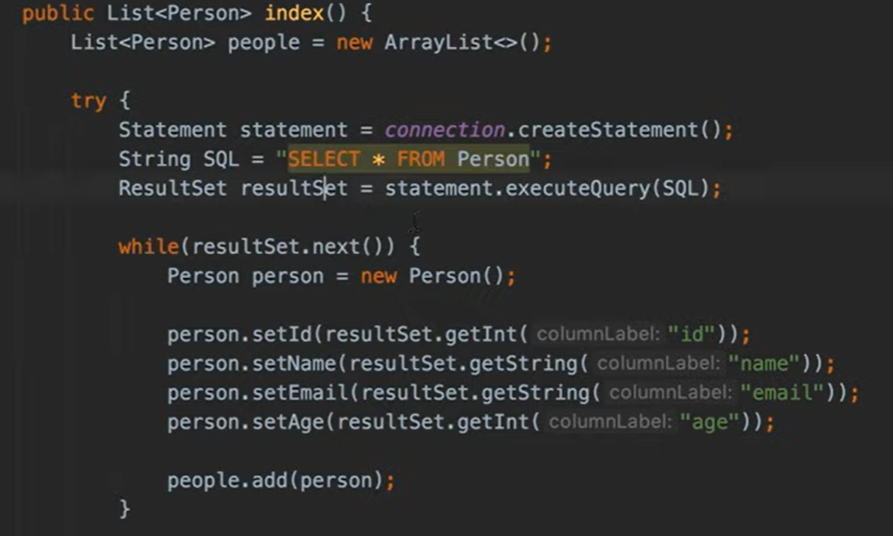




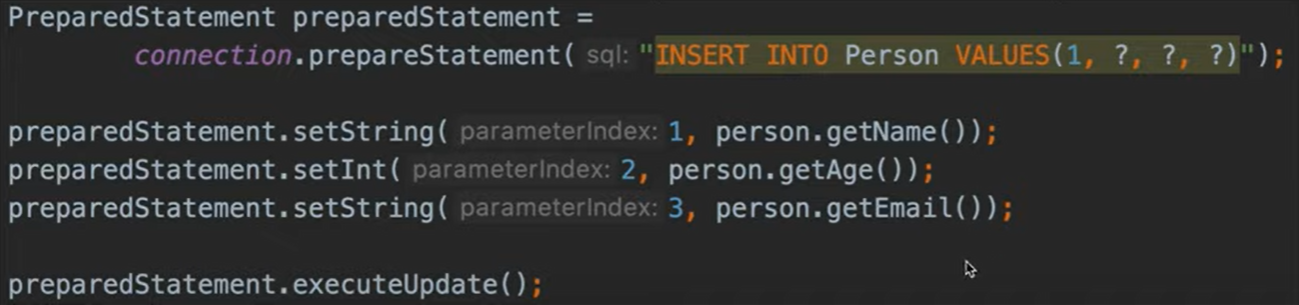
Рассмотрим подключение к БД через **JDBC**. Для подключения к БД нужно подключить зависимость для работы с JDBC и настроить параметры подключения:



И создадим запрос в БД:

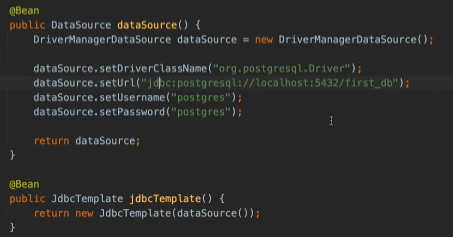


Чтобы защититься от SQL-инъекций, для всех запросов с данными, полученными от пользователя, используется **PreparedStatement**. В этом случае SQL запрос не сможет быть изменен, кроме вставляемых данных в подготовленные места.

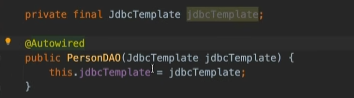


PreparedStatement в отличии от Statement компилируется один раз. В связи с этим он быстрее работает. Дополнительно он кэшируется на БД.

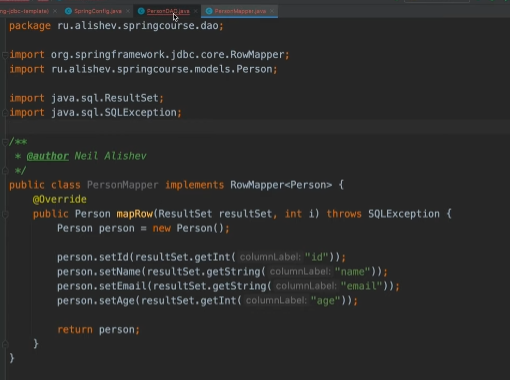
Рассмотрим подключение к БД через **JdbcTemplate**. Для подключения к БД нужно подключить зависимость для работы с JdbcTemplate и создать бины JdbcTemplate и DataSource с настройками подключения:



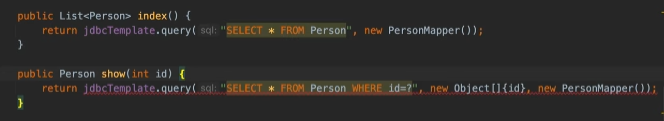
Внедрить бин JdbcTemplate в экземпляр класса, отвечающего за доступ к БД:



Создать класс, реализующий интерфейс RowMapper, для преобразования возвращаемой из БД строки в объект java класса:



И создать запрос к БД:



В JdbcTemplate по умолчанию используется PrepareStatement.

Также можно использовать уже готовый шаблон RowMapper (когда имена полей класса и таблицы совпадают). В этом случае запрос к БД будет таким:

