БелорусскиЙ государственный университет

информатики и радиоэлектроники

**Отчет по лабораторной работе №5-6**

«**JSF, EJB, JPA**»

|  |  |  |
| --- | --- | --- |
| Выполнила студентка: |  | Назарчук Дина  гр. 272301 |
|  | | |

Минск 2015

***Цель:***

Познакомиться с JSF, EJB, JPA. Разобраться как сконфигурировать JPA.

***Задания:***

1.Запустит пример  <https://www.youtube.com/watch?v=QIGTie-ULic>

2. Пояснить как сконфигурирован JPA.

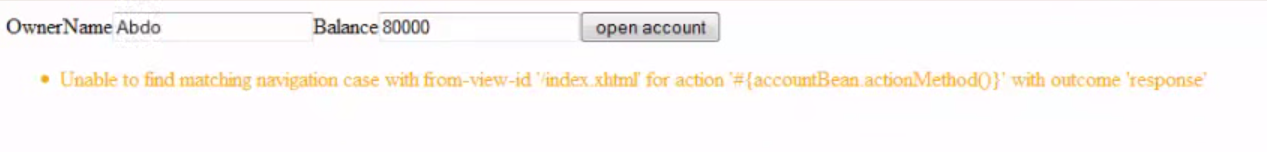
3. Объяснить чем отличается JSF от JSP.

4. Объяснить что такое dependency injection и inversion of control.

5.Пояснить в чем отличие stateless от statfull beans.

***Реализация :***

**1. Результат запуска примера**:



**2. Как сконфигурирован JPA :**

Необходимо создать так называемый класс-сущность (Entity). По сути, Entity – обычный Java-класс, который должен удовлетворять нескольким требованиям. Создадим наш Entity. Кликните на созданном проекте правой кнопкой мыши → New → Entity Class... В появившемся диалоговом окне укажите название класса-сущности и пакет, где он будет располагаться.

**3. Отличие JSF от JSP**:

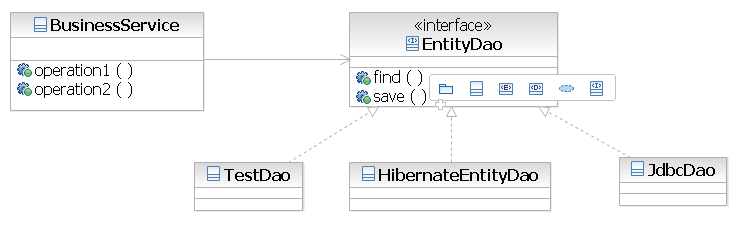
JSF - это стандартный Sun/Oracle фреймворк реализующий паттерн MVC, где JSP используется в качестве компонента View

# JSP использует сервлеты и в результате компиляции превращается в сервлет. JSF использует сервлеты и JSP - это фреймворк поверх них.

**4. Что такое** **dependency injection и inversion of control :**

***Dependency Injection*** *(DI)*, называемый также часто ***Inversion of Control*** *(IoC)* (в русскоязычном мире также используется термин *внедрение зависимости*), является одним из основополагающих принципов современных контейнеров программного обеспечения.

В традиционных системах компонент получает прямую ссылку на местонахождение необходимых для работы объектов или сервисов, или обращается к [сервис-локатору](http://java.sun.com/blueprints/corej2eepatterns/Patterns/ServiceLocator.html)и запрашивает ссылку на реализацию определенного типа сервиса, то в случае DI контейнера окружение на основе конфигурационных данных, которые могут быть предоставлены системы в виде xml-файла или аннотаций, само знает о необходимых взаимосвязях между компонентами, и предоставляет нужные объекты во время инициализации или исполнения. Использование внедрение зависимости вносит дополнительную гибкость в систему, поскольку облегчается создание альтернативных реализаций сервиса, и позволяет указывать в конфигурации, какая именно реализация должна быть использована.



**Рисунок 1. Пример использования Dependency Injection**

На рисунке 1 приведен пример диаграммы классов, отражающую зависимость, которую можно встретить практически в любой реальной системе. Как следует из нее, сервис BusinessService использует для работы с базой данных интерфейс EntityDao и существует несколько реализаций этого интерфейса: TestDao, HibernateDao и JdbcDao. TestDao используется во время тестирования сервиса, а выбор между HibernateDao и JdbcDao желательно должен осуществляет администратор системы основываясь на своих предпочтениях или объективных показателях производительности. В случае, когда BusinessService сам создает экземпляр класса реализующего интерфейс EntityDao, разработка, а главное развитие и сопровождение такой системы значительно усложняется, поскольку тогда для тестирования необходимо создать экземпляр TestDao, потом после успешного тестирования изменить код сервиса BusinessService, и затем перекомпилировать его, что само по себе может привнести новую ошибку. Использование JNDI внутри BusinessService (такой подход также называется Dependency Lookup) решает вопрос с перекомпиляцией после тестирования системы. Такой способ был основным во времена EJB2 контейнеров, которые не поддерживают принцип конфигурирования зависимостей. Компоненту необходимо самому спрашивать у контейнера, где находятся необходимые для работы объекты. Однако при этом увеличивается объем служебного кода в системе (glue code), то есть кода, который не был разработан для решения бизнес требований системы, а для связывания воедино различных частей системы, обработки исключительных ситуаций и т.д. Кроме этого, тестирование BusinessService при таком подходе возможно только внутри EJB2 контейнера, что делает организацию unit tests совсем не тривиальной задачей. Используемый часто в таких контейнерах паттерн [ServiceLocator](http://java.sun.com/blueprints/corej2eepatterns/Patterns/ServiceLocator.html), который инкапсулирует внутри все JNDI-вызовы, также позволяет не раздувать размеры вспомогательного кода, однако соответствующий метод ServiceLocator все равно должен быть вызван явно внутри BusinessService.

Наиболее красивым и эффективным решением является использование принципа конфигурирования зависимостей (dependency injection) и контейнера, поддерживающего этот принцип. В этом случае экземпляр класса, реализующий интерфейс EntityDao, инициализируется внутри контейнера на основании указанной администратором конфигурации. После этого контейнером ссылка на созданную реализацию EntityDao автоматически «впрыскивается» в соответствующее поле класса BusinessService.

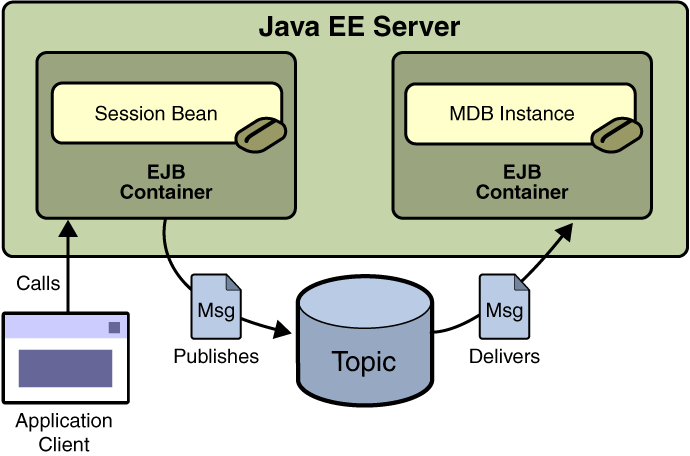
Основные преимущества использования контейнера, реализующего принцип внедрения зависимостей:

* Централизованное и внешнее управление зависимостями, в случае использования текстовых конфигурационных файлов, которые не подлежат компиляции, и может быть адаптировано индивидуально к различным стадиям разработки и функционирования системы, не требуя при этом перекомпиляции. При традиционном подходе необходимые компоненты инициализируются либо по мере их необходимости – внутри использующих их классов, или же используются фабрики классов (class factory), что тоже не является идеальным способом, поскольку тогда использующий класс получает также зависимость от этих фабрик.
* Отсутствие вспомогательного связующего кода. При использовании контейнера, поддерживающего принцип DI, значительно уменьшается размер кода, поскольку все необходимые зависимости разрешается и компоненты «склеиваются» автоматически.
* Улучшение тестируемости отдельных компонент системы. Использование dependency injection позволяет легко заменять зависимые компоненты, что является особенно полезным для организации unit test системы. Рассмотрим приведенный выше пример, показанный на рисунке 1. Для того, чтобы протестировать реализацию BusinessService, надо проверить правильность работы с различными данными, которые этот сервис получает используя интерфейс EntityDao. Для этого можно использовать экземпляр объекта TestDao, имитирующего работу с базой данной и предоставляющего сервису необходимые тестовые данные. Проверку же правильности реализации JdbcDao или HibernateEntityDao можно провести в другом тесте.
* Использование лучших методик разработки программного обеспечения. Создание приложений с использованием DI контейнера подталкивает разработчика к интенсивному использованию интерфейсов. Обычно все основные компоненты определяются с помощью интерфейсов, потом создается конкретные реализации этих интерфейсов, которые связываются с помощью DI. Собственно такого рода приложения в платформе Java были возможны до появления dependency injection и DI контейнеров. Но использование Spring или аналогичного ему контейнера позволяет использовать проверенную реализацию паттерна DI, позволяя разработчикам концентрироваться на создании бизнес логики.

**5. Отличие stateless от statfull beans:**

**Session beans**- используется для построения бизнес-логики, которая может быть вызвана программным клиентом через локальный, удаленный или веб-интерфейс обслуживания клиентов.

Для доступа к приложению, развернутого на сервере, клиент вызывает методы сессионного компонента. Сессионный компонент выполняет работу для своего клиента, защищая его от сложности, выполняя бизнес-задач внутри сервера.



Существует 2 типа session-beans: **stateless** и **stateful**.

**Stateful**- автоматически сохраняют свое состояние между разными клиентскими вызовами.

**Stateless**- используются для реализации **бизнесс-процессов**, которые могут быть завершены за одну операцию.