**Трофеи с отчетов к 3ей лабе**

**Hibernate session interface**

Интерфейс org.hibernate.Session является мостом между приложением и Hibernate. С помощью сессий выполняются все CRUD-операции с объектами-сущностями. Объект типа Session получают из экземпляра типа org.hibernate.SessionFactory, который должен присутствовать в приложении в виде singleton. Жизненный цикл session ограничен началом и концом логической транзакции.

**Состояния объектов-сущностей**

* Transient object -- заполненные экземпляры классов-сущностей. Могут быть сохранены в БД. Не присоединены к сессии. Поле Id не должно быть заполнено, иначе объект имеет статус detached.
* Persistent object -- так называемая хранимая сущность, которая присоединена к конкретной сессии. Только в этом статусе объект взаимодействует с базой данных. При работе с объектом данного типа в рамках транзакции все изменения объекта записываются в базу.
* Detached object -- объект, отсоединённый от сессии, может существовать или не существовать в БД.

**Некоторые методы из интерфейса Session**

Любой объект-сущность можно переводить из одного статуса в другой. Для этого в интерфейсе Session существуют следующие методы:

* persist(Object) -- преобразует объект из *transient* в *persistent*, то есть присоединяет к сессии и сохраняет в БД. Однако, если мы присвоим значение полю Id объекта, то получим PersistentObjectException, т.к. Hibernate посчитает, что объект *detached*, т.е. существует в БД. При сохранении метод *persist()* сразу выполняет *insert*, не делая *select*.
* merge(Object) -- преобразует объект из *transient* или *detached* в persistent. Если из *transient*, то работает аналогично *persist()* (генерирует для объекта новый Id, даже если он задан), если из *detached* — загружает объект из БД, присоединяет к сессии, а при сохранении выполняет запрос *update*.
* replicate(Object, ReplicationMode) -- преобразует объект из *detached* в *persistent*, при этом у объекта обязательно должен быть заранее установлен Id. Данный метод предназначен для сохранения в БД объекта с заданным Id, чего не позволяют сделать *persist()* и *merge()*. Если объект с данным Id уже существует в БД, то поведение определяется согласно правилу из перечисления org.hibernate.ReplicationMode:
  + ReplicationMode.IGNORE — ничего не меняется в базе.
  + ReplicationMode.OVERWRITE — объект сохраняется в базу вместо существующего.
  + ReplicationMode.LATEST\_VERSION — в базе сохраняется объект с последней версией.
  + ReplicationMode.EXCEPTION — генерирует исключение.
* delete(Object) -- удаляет объект из БД, иными словами, преобразует *persistent* в *transient*. Object может быть в любом статусе, главное, чтобы был установлен Id.
* save(Object) -- сохраняет объект в БД, генерируя новый Id, даже если он установлен. Object может быть в статусе *transient* или *detached*.
* update(Object) -- обновляет объект в БД, преобразуя его в *persistent* (Object в статусе *detached*).
* saveOrUpdate(Object) -- вызывает *save()* или *update()*.
* refresh(Object) -- обновляет *detached*-объект, выполнив *select* к БД, и преобразует его в *persistent*.
* get(Object.class, id) -- получает из БД объект класса-сущности с определённым Id в статусе *persistent*.

Объект Session кэширует у себя загруженные объекты, при загрузке объекта из БД в первую очередь проверяется кэш. Для того, чтобы удалить объект из кэша и отсоединить от сессии, используется session.evict(Object). Метод session.clear() применит evict() ко всем объектам в сессии.

**EntityManager**

Является интерфейсом ORM, служит для управления персистентными сущностями.

Управление сущностями начинается с создания EntityManagerFactory, которая отвечает за отображение объектов в базу, поддержку соединений, кэш состояний и всякие такие вещи.

Фабрика создает объекты EntityManager, который может управлять сущнастями. EntityManager может образовать *persistence context* - набор экземпляров сущностей, загруженных из БД или созданных (является своего рода кэшем данных в рамках транзакции). EntityManager сбрасывает в БД все изменения в персистентном контексте в момент коммита транзакции, либо при явном вызове метода flush().

* persist() -- вводит новый экземпляр managed сущности в persistence context. При коммите транзакции командой SQL INSERT в БД будет создана соответствующая запись.
* merge() -- переносит состояние detached сущности в persistence context: из БД загружается экземпляр с тем же id, в него переносится состояние переданного Detached экземпляра и возвращается загруженный Managed экземпляр. Далее надо работать именно с возвращенным Managed экземпляром.
* remove() -- удаляет объект из бд, либо, если включен режим мягкого удаления, установит атрибуты deleteTs и deletedBy.
* find() -- загружает экземпляр сущности по идентификатору.
* fetch() -- обеспечивает для экземпляра сущности загрузку всех атрибутов указанного представления, включая lazy атрибуты. Экземпляр сущности должен быть в Managed состоянии.
* reload() -- перезагрузить экземпляр сущности с указанным представлением. Обеспечивает загрузку всех атрибутов представления, вызывая внутри себя метод fetch().

**Вопросы с se.ifmo**

**1. Технология JavaServer Faces. Особенности, отличия от сервлетов и JSP, преимущества и недостатки. Структура JSF-приложения.**

**JavaServer Faces (JSF)** — это фреймворк для веб-приложений, для разработки пользовательских интерфейсов Java EE приложений. Основывается на использовании компонентов. Состояние компонентов пользовательского интерфейса сохраняется, когда пользователь запрашивает новую страницу и затем восстанавливается, если запрос повторяется.

**Преимущества JSF**

* Четкое разделение бизнес-логики и интерфейса
* Управление на уровне компонент
* Простая работа с событиями на стороне сервера
* Расширяемость
* Доступность нескольких реализаций от различных компаний-разработчиков
* Широкая поддержка со стороны интегрированных средств разработки (IDE)

**Недостатки JSF**

* Высокоуровневый фреймворк — сложно реализовывать не предусмотренную авторами функциональность.
* Сложности с обработкой GET-запросов (устранены в JSF 2.0).
* Сложность разработки собственных компонентов.

**Структура JSF-приложения**

* JSP-страницы с компонентами GUI
* Библиотека тегов
* Управляемые бины
* Доп. объекты(компоненты, конвертеры, вылидаторы)
* Доп. теги
* Конфигурация – faces-config.xml
* Дискриптор развертывания – web.xml

**2. Использование JSP-страниц и Facelets-шаблонов в JSF-приложениях.**

Интерфейс JSF-приложения состоит из страниц JSP (Java Server Pages), которые содержат компоненты, обеспечивающие функциональность интерфейса. При этом библиотеки тегов JSP используются на JSF-страницах для отрисовки компонентов интерфейса, регистрации обработчиков событий, связывания компонентов с валидаторами и конвертаторами данных и много другого.

При этом нельзя сказать, что JSF неразрывно связана с JSP, т.к. теги, используемые на JSP-страницах только отрисовывают компоненты, обращаясь к ним по имени. Жизненный же цикл компонентов JSF не ограничивается JSP-страницей.

**3. JSF-компоненты - особенности реализации, иерархия классов. Дополнительные библиотеки компонентов. Модель обработки событий в JSF-приложениях.**

**Особенности реализации JSF-компонент**

* Интерфейс строится из компонентов.
* Компоненты расположены на страницах JSP.
* Компоненты реализуют интерфейс javax.faces.component.UIComponent.
* Можно создавать собственные компоненты.
* Компоненты на странице объединены в древовидную структуру — представление.
* Корневым элементов представления является экземпляр класса javax.faces.component.UIViewRoot.

Некоторые компоненты JSF: <f:subview>, <h:selectOneMenu>, <h:selectOneRadio>, <h:selectOneListbox>, <h:selectManyCheckbox>, <selectManyListbox>, <selectManyMenu>, <h:textArea>, ...

<h:selectOneListbox id="type" value="#{contactController.contact.type}">

<f:selectItem itemValue="PERSONAL" itemLabel="personal"/>

<f:selectItem itemValue="BUSINESS" itemLabel="business"/>

</h:selectOneListbox>

**Иерархия классов (фрагмент)**

-- javax.faces.component.UIComponent

---- javax.faces.component.UIComponentBase

------ javax.faces.component.UIOutput

-------- javax.faces.component.UIInput

---------- javax.faces.component.UISelectOne

---------- javax.faces.component.UISelectMany

**Дополнительные библиотеки компонентов.**

[PrimeFaces](https://www.primefaces.org/showcase/), [RichFaces](http://docs.jboss.org/richfaces/latest_4_5_X/Developer_Guide/en-US/html/appe-Developer_Guide-Style_classes_and_skin_parameters.html), [ICEFaces](http://www.icesoft.org/java/projects/ICEfaces/ace-components.jsf), OpenFaces, Trinidad, Tomahawk.

**Модель обработки событий**

**Жизненный цикл обработки запроса в приложениях JSF состоит из следующих фаз:**

1. Восстановление представления
2. Использование параметров запроса; обработка событий
3. Проверка данных; обработка событий
4. Обновление данных модели; обработка событий
5. Вызов приложения; обработка событий

**Вывод результата**

1. Фаза формирования представления. JSF Runtime формирует представление по запросу(request) пользователя: создаются объекты компонентов, назначаются слушатели событий, конвертеры и валидаторы, все элементы представления помещаются в FacesContext
2. Фаза получения значений компонентов. Вызывается конвертер из стокового типа данных в требуемый тип. Если конвертация успешна, то значение сохраняется в локальной переменной компонента. Если неуспешно – создается сообщение об ошибке и помещается в FacesContext.
3. Фаза валидации значений компонентов. Вызываются валидаторы, зарегистрированные для компонентов представления. Если значение компонента не проходит валидацию, создается сообщение об ошибке и сохраняется в FacesContext.
4. Фаза обновления значений компонентов. Если данные валидны, то значение компонента обновляется. Новое значение присваивается полю объекта компонента.
5. Фаза вызова приложения. Управление передается слушателям событий. Формируются новые значение компонентов.
6. Фаза формирования ответа сервера. Обновляется представление в соответствии с результатом обработки запроса. Если это первый запрос к странице, то компоненты помещаются в иерархию представления. Формируется ответ сервера на запрос(response). На стороне клиента происходит обновление страницы.

**4. Конвертеры и валидаторы данных.**

JSF имеет встроенные конвенторы и позволяет создавать специализированные.

**Стандартные конвертеры JSF**

* javax.faces.BigDecimal
* javax.faces.BigInteger
* javax.faces.Boolean
* javax.faces.Byte
* javax.faces.Character
* javax.faces.DateTime
* javax.faces.Double
* javax.faces.Float

<h:outputLabel value="Age" for="age" accesskey="age" />

<h:inputText id="age" size="3" value="#{contactController.contact.age}">

</h:inputText>

<h:outputLabel value="Birth Date" for="birthDate" accesskey="b" />

<h:inputText id="birthDate" value="#{contactController.contact.birthDate}">

<f:convertDateTime pattern="MM/yyyy"/>

</h:inputText>

**Cпециализированные конвертеры**

1. Создать класс, реализующий интерфейс Converter
2. Реализовать метод getAsObject(), для преобразования строкового значения поля в объект.
3. Реализовать метод getAsString.
4. Зарегистрировать конвертер в контексте Faces в файле faces-config.xml, используя элемент .

**файл faces-config.xml**

<converter>

<converter-for-class>

com.arcmind.contact.model.Group

</converter-for-class>

<converter-class>

com.arcmind.contact.converter.GroupConverter (com.arcmind.contact.converter.TagConverter)

</converter-class>

</converter>

**Валидаторы**

**Существует 4 типа валидаторов**

1. С помощью встроенных компонентов
2. На уровне приложения
3. С помощью проверочных методов серверных объектов (inline-валидация)
4. С помощью специализированных компонентов, реализующих интерфейс Validator

**1. С помощью встроенных компонентов**

1. DoubleRangeValidator
2. LongRangeValidator
3. LengthValidator

<%-- возраст (age) --%>

<h:outputLabel value="Age" for="age" accesskey="age" />

<h:inputText id="age" size="3" value="#{contactController.contact.age}">

<f:validateLongRange minimum="0" maximum="150"/>

</h:inputText>

<h:message for="age" errorClass="errorClass" />

**2. На уровне приложения**

Это непосредственно бизнес-логика. Заключается в добавлении в методы управляемых bean-объектов кода, который использует модель приложения для проверки уже помещенных в нее данных.

**3. С помощью проверочных методов серверных объектов**

Для типов данных, не поддерживаемых стандартными валидаторами, например, адресов электронной почты, можно создавать собственные валидирующие компоненты

**4. С помощью специализированных компонентов, реализующих интерфейс Validator**

JSF позволяет создавать подключаемые валидирующие компоненты, которые можно использовать в различных Web-приложениях.

Это должен быть класс, реализующий интерфейс Validator, в котором реальзован метод validate(). Необходимо зарегистрировать валидатор в файле faces-config.xml. После этого можно использовать тег <f:validator/> на страницах JSP.

**faces-config.xml**

<validator>

<validator-id>arcmind.zipCode</validator-id>

<validator-class>com.arcmind.validators.ZipCodeValidator</validator-class>

</validator>

**5. Представление страницы JSF на стороне сервера. Класс UIViewRoot.**

За представление отвечают:

UI Component. Объект с состоянием, методами, событиями, который содержится на сервере и отвечает за взаимодействие с пользователем (визуальный компонент). Каждый UI компонент содержит метод метод render для прорисовки самого себя, согдасно правилам в классе Render

Renderer - Отвечает за отображение компонента и преобразование ввода пользователя

Validator, Convertor

Backing bean - собирает значения из компонент, реагирует на события, взаимодействует с бизнес-логикой.

Events, Listeners, Message

Navigation - =правила навигации между страницами, задаются в виде xml документа

**UIViewRoot**

Объект UIViewRoot дает представдение JSF, он связан с активным FacesContext. JSF реализация создаёт представление при первом обращении (запросе), либо восстанавливает уже созданное. Когда клиент отправляет форму (postback), JSF конвертирует отправленные данные, проверяет их, сохраняет в managed bean, находит представление для навигации, восстанавливает значения компонента из managed bean, генерирует ответ по представлению. Все эти действия JSF описываются с помощью 6 упорядоченных процессов.

**6. Управляемые бины - назначение, способы конфигурации. Контекст управляемых бинов.**

**Управляемые бины** – классы, содержащие параметры и методы для обработки данных с компонентов. Должны иметь методы get и set/ Используются для обработки UI и валидации данных. Жихненным цикллом управляет JSF Runtime Env. Доступ из JSP-страниц осуществляется с помощью языка выражений (EL). Конфигурация задается либо в faces-config.xml, либо с помощью аннотаций.

**Конфиуграция управляемых бинов**

**faces-config.xml**

<managed-bean>

<managed-bean-name>customer</managed-bean-name>

<managed-bean-class>CustomerBean</managed-bean-class>

<managed-bean-scope>request</managed-bean-scope>

<managed-property>

<property-name>areaCode</property-name>

<value>#{initParam.defaultAreaCode}</value>

</managed-property>

</managed-bean>

**С помощью аннотаций**

@ManagedBean(name="customer")

@RequestScoped

public class CustomerBean {

@ManagedProperty(value="#{initParam.defaultAreaCode}" name="areaCode")

private String areaCode;

...

}

Managed bean - бин, зарегистрированный в JSF, управляется JSF платформой. Мanaged bean используются в качестве модели для компонентов и имеют свою область жизни (scope), которую можно задать при помощи аннотации или в конфигурационном файле faces-config.xml.

У управляемых бинов есть **контекст**, который оперделяет продолжительность жизни. Он задается аннотацией.

**Аннотации**

@RequestScoped - используется по умолчанию. Создаётся новый экземпляр managed bean на каждый HTTP запрос (и при отправке, и при получении). *Контекст - запрос*

@SessionScoped - экземпляр создаётся один раз при обращении пользователя к приложению, и используется на протяжении жизни сессии. Managed bean обязательно должен быть Serializable. *Контекст — сессия*.

@ApplicationScoped - экземпляр создаётся один раз при обращении и используется на протяжении жизни всего приложения. Не должен иметь состояния, а если имеет, то должен синхронизировать доступ, так как доступен для всех пользователей. *Контекст — приложение.*

@ViewScoped - экземпляр создаётся один раз при обращении к странице, и используется ровно столько, сколько пользователь находится на странице (включая ajax запросы). *Контекст — страница, представление*.

@CustomScoped(value="#{someMap}") - экземпляр создаётся и сохраняется в Map. Программист сам управляет областью жизни.

@NoneScoped - экземпялр создаётся, но не привязывается ни к одной области жизни. Применяется когда к нему обращаются другие managed bean'ы, имеющие область жизни. *Бин без контекста.*

**7. Конфигурация JSF-приложений. Файл faces-config.xml. Класс FacesServlet.**

**faces-config.xml** — конфигурационный файл JavaServer Faces, который должен находиться в директории WEB-INF проекта. В этом файле могут находиться настройки managed bean, конвертеры, валидаторы, локализация, навигации и другие настройки, связанные с JSF

**faces-config.xml**

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

<faces-config xmlns="http://java.sun.com/xml/ns/javaee"

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

xsi:schemaLocation="http://java.sun.com/xml/ns/javaee

http://java.sun.com/xml/ns/javaee/web-facesconfig\_1\_2.xsd"

version="1.2">

<managed-bean>

<managed-bean-name>calculator</managed-bean-name>

<managed-bean-class>com.arcmind.jsfquickstart.model.Calculator</managed-bean-class>

<managed-bean-scope>request</managed-bean-scope>

</managed-bean>

</faces-config>

Объявление управляемого объекта: имя объекта задается с помощью <managed-bean-name>, полное имя класса - <managed-bean-class>. Класс управляемого объекта обязан содержать конструктор без параметров.

<managed-bean-scope> определяет, где JSF будет искать объект. Если объект привязан к представлению и не существует на момент обращения, то JSF создаст его автоматически с помощью API универсального языка выражений EL. Объект будет доступен в течение обработки одного запроса.

По умолчанию используется faces-config, но можно использовать дополнительные конфиги, перечислив их в web.xml.

**Класс FacesServlet**

* Обрабатывает запросы с браузера.
* Формирует объекты-события и вызывает методы-слушатели.

**8. Навигация в JSF-приложениях.**

Механизм нафигации JSF позволяет определить связь между логическим признаком результата и следующим представлением. Реализуется объектами NavigationHandler. Навигация осуществляется с помощью правил перехода.

Ссылку можно добавить тремя различными способами:

1. С помощью commandLink и обычного правила перехода, определяемого в faces-config.xml

<navigation-rule>

<navigation-case>

<from-outcome>CALCULATOR</from-outcome>

<to-view-id>/pages/calculator.jsp</to-view-id>

</navigation-case>

</navigation-rule>

1. С помощью commandLink и правила перехода, использующего элемент .
2. Связывание с помощью прямой ссылки (элемента <h:outputLink>)

<h:outputLink value="pages/calculator.jsf">

<h:outputText value="Calculator Application (outputlink)"/>

</h:outputLink>

**9. Доступ к БД из Java-приложений. Протокол JDBC, формирование запросов, работа с драйверами СУБД.**

**Протокол JDBC:**

JDBC (Java Database Connectivity) является не протоколом, а интерфейсом и основан на интерфейсе уровня вызова группы доступа SQL. Сам по себе JDBC работать не может и использует основные абстракции и методы ODBC. Хотя в стандарте JDBC API и предусмотрена возможность работы не только через ODBC, а и через использование прямых линков к базам данных по двух- или трех-звенной схеме, эту схему используют гораздо реже, чем повсеместно используемый JDBC-ODBC-Bridge занимающий центральное место в общей схеме взаимодействия интерфейсов.

**Запрос:**

Запрос (query) – это средство выбора необходимой информации из базы данных. Применяются два типа запросов: по образцу (QBE) и структурированный язык запросов (SQL). Существует несколько типов запросов: на выборку, на обновление, на добавление, на удаление, перекрестный запрос, создание таблиц. Запросы на выборку используются для отбора нужной пользователю информации, содержащейся в таблицах, используются чаще всего. Они создаются только для связанных таблиц.

**Драйвер СУБД:**

Есть такая вещь как передаваемые в запрос параметры и самое главное - результат выполнения запроса. Задача драйвера упаковывать запрос, параметры и результаты в пакеты передаваемые по сети. И формат упаковки языком никак не оговаривается и каждая СУБД реализует его по своему. Тем более, что во многих СУБД существуют не только разные типы данных, но и сложные типы, например, как массивы PostgreSQL, которых в стандартах SQL вообще нет, поэтому для работы с СУБД программы линкуются с библиотеками конкретных СУБД, которые умеют общаться с конкретной БД.

**10. Концепция ORM. Библиотеки ORM в приложениях на Java. Основные API. Интеграция ORM-провайдеров с драйверами JDBC.**

**Концепция ORM:**

Реляционной СУБД –хранит данные в виде привычных всем таблиц, применяются так широко вовсе потому что они надёжны, быстры и, самое главное, привычны. Объектно-ориентированные СУБД - хранит данные в виде точно таких же объектов, какими оперируют программисты в своём коде. ORM применение специальных фреймворков или библиотек, которые сами занимаются связыванием объектов в программе и записей в таблицах базы данных(мы используем ORM потому что пишем на java, а там оперируют объектами, но взаимодействовать нам надо с реляционной СУБД) Библиотеки ORM в java: ActiveJDBC,EclipseLink,Hibernate,Java Persistence API

**Основные API:**

Java Persistence API (JPA) — спецификация API Java EE, предоставляет возможность сохранять в удобном виде Java-объекты в базе данных, реализует концепцию ORM. Существует несколько реализаций этого интерфейса, одна из самых популярных использует для этого Hibernate.

**11. Библиотеки ORM Hibernate и EclipseLink. Особенности, API, сходства и отличия.**

**EclipseLin:**

EclipseLin предоставляет реализацию JPA с открытым кодом. Кроме того, EclipseLink поддерживает ряд других стандартов сохраняемости, таких как Java Architecture for XML Binding (Проще говоря, вместо того, чтобы сохранять объект в строке базы данных, JAXB отображает его в XML-представление). Одним из основных преимуществ EclipseLink является то, что вы можете вызывать собственные функции SQL непосредственно в своих запросах JPQL. В Hibernate это невозможно напрямую.. EclipseLink предлагает другие параметры, которых нет в Hibernate. Например:

* @ReadOnly - указывает, что сущность для сохранения доступна только для чтения
* @Struct - определяет класс для сопоставления с типом структуры базы данных.

**Hibernate:**

Но Hibernate имеет лучшую документацию, а также лучшие сообщения об ошибках.Например, давайте посмотрим на несколько аннотаций, предлагаемых Hibernate, которые расширяют функциональность @Entity:

* @Table - позволяет указать имя таблицы, созданной для сущности
* @BatchSize - указывает размер пакета при извлечении сущностей из таблицы Также стоит отметить несколько дополнительных функций, которые не указаны в JPA, которые могут оказаться полезными в более крупных приложениях:
* Настраиваемые операторы CRUD с аннотациями @SQLInsert, @SQLUpate и @SQLDelete
* Неизменяемые сущности с аннотацией @Immutable

Также стоит сказать, что EclipseLink больше соответствует стандартам, поскольку это эталонная реализация для JPA 2, Hibernate имеет некоторые проблемы с совместимостью, но он более зрелый.

**12. Технология JPA. Особенности, API, интеграция с ORM-провайдерами.**

JPA – это технология, обеспечивающая объектно-реляционное отображение простых JAVA объектов и предоставляющая API для сохранения, получения и управления такими объектами. Сам JPA не умеет ни сохранять, ни управлять объектами, JPA только определяет правила игры: как что-то будет действовать. JPA также определяет интерфейсы, которые должны будут быть реализованы провайдерами. Плюс к этому JPA определяет правила о том, как должны описываться метаданные отображения и о том, как должны работать провайдеры. Дальше, каждый провайдер, реализуя JPA определяет получение, сохранение и управление объектами. У каждого провайдера реализация разная.

Поддержка сохранности данных, предоставляемая JPA, покрывает области:

* непосредственно API, заданный в пакете javax.persistence;
* платформо-независимый объектно-ориентированный язык запросов Java Persistence Query Language;
* метаинформация, описывающая связи между объектами.
* Генерация DDL для сущностей