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

**JavaServer Faces (JSF)** — это фреймворк для веб-приложений, написанный на Java. Он служит для того, чтобы облегчать разработку пользовательских интерфейсов для Java EE приложений. В отличие от прочих MVC фреймворков, которые управляются запросами, подход JSF основывается на использовании компонентов. Состояние компонентов пользовательского интерфейса сохраняется, когда пользователь запрашивает новую страницу и затем восстанавливается, если запрос повторяется. Для отображения данных обычно используется JSP, но JSF можно приспособить и под другие технологии, например XUL.

// WTF??1?!!??! **Технология JavaServer Faces включает:**

* Набор API для представления компонент пользовательского интерфейса (UI) и управления их состоянием, обработкой событий и валидацией вводимой информации, определения навигации, а также поддержку интернационализации (i18n) и доступности (accessibility).
* Специальная библиотека JSP тегов для выражения интерфейса JSF на JSP странице.
* Призванная быть гибкой, технология JavaServer Faces усиливает существующие, стандартные концепции пользовательского интерфейса (UI) и концепции Web-уровня без привязки разработчика к конкретному языку разметки, протоколу или клиентскому устройству.

**Особенности, отличия от сервлетов и JSP**

* JSF предлагает компоненто-центрический, независимый от клиента подход к разработке пользовательских интерфейсов для Web, таким образом увеличивая продуктивность разработчика и простой в использовании;
* Упрощает доступ и управление данными приложения через Web-интерфейс;
* Автоматически управляет состоянием пользовательского интерфейса между несколькими запросами и несколькими клиентами простым и ненавязчивым способом;

| **jsf** | **jsp** |
| --- | --- |
| jsf is a web application that is used to simplify development integration of web based user interfaces. | jsp is a java based technology used specifically in order to help software developers create dynamic web pages. |
| jsf contains multiple core features, including, but not limited to, managed beans, a template based component system, and two xml based tag libraries. | jsp must be compiled in java bytecode in order to function properly.jsp must be compiled in java bytecode in order to function properly. |
| jsf is a framework. model:backing bean view:jsf controller:facesservlet | jsp is not a request driven model view controller, but jsp is accessed by the dynamically generated web pages like html or xml. |
| jsf supports validator and conversion,ajax. | jsp does not. |

**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)
* упрощает обмен данными между UI компонентами
* позволяет реализовывать собственные компоненты
* обеспечивает UI компоненты, которые могут быть повторно использованы
* управляет состоянием UI, которое зависит от нескольких серверов
* соединяет события на стороне клиента с серверной частью приложения

**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 или XHTML-страницы, содержащие компоненты GUI.
* Библиотеки тегов.
* Управляемые бины.
* Дополнительные объекты (компоненты, конвертеры и валидаторы).
* Дополнительные теги.
* Конфигурация — faces-confg.xml (опционально).
* Дескриптор развёртывания — web.xml.

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

JSP-это старая технология просмотра и широко используется в сочетании с JSF 1.x. Facelets (некоторые люди чрезмерно обобщены как XHTML) является преемником JSP и введена в качестве технологии просмотра по умолчанию JSF 2.x в конце 2009 года

Конечно, с JSF 2.0 контейнер по умолчанию будет искать файлы xhtml, что не очень хорошо, если вы все еще работаете с JavaServer Pages. Но не волнуйтесь, это простое решение. Все, что вам нужно сделать, это отредактировать файл web.xml и сообщить контейнеру, что вашей технологией представления по умолчанию должны быть JSP.

<context-param>

<param-name>javax.faces.DEFAULT\_SUFFIX</param-name>

<param-value>.jsp</param-value>

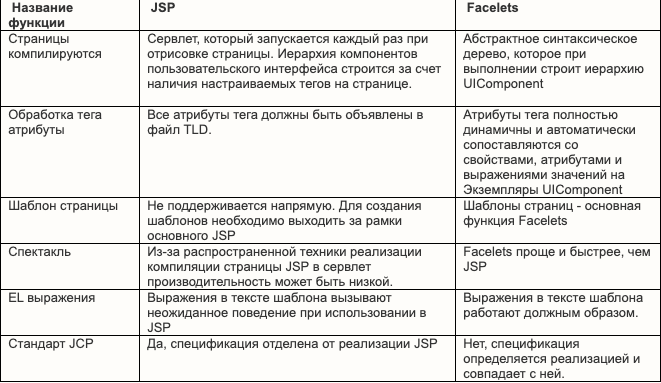
</context-param>

JSF представляет собой структуру MVC на основе компонентов, которая построена поверх API Servlet и предоставляет компоненты через taglibs который может использоваться в JSP или любой другой технологии просмотра на основе Java, такой как Facelets. Facelets гораздо более подходит для JSF, чем JSP. Это обеспечивает отличные возможности шаблонов, такие как составные компоненты, в то время как JSP в основном предлагает jsp:include для шаблонов, так что вы вынуждены создавать пользовательские компоненты с необработанным кодом Java (который немного непрозрачен и много утомительной работы в JSF), когда вы хотите заменить повторяющуюся группу компонентов одним компонентом. Начиная с JSF 2.0, JSP устарела как технология просмотра в пользу Facelets.

Facelets — открытый веб-фреймворк, распространяемый под лицензией Apache license. Технология управления представлением для JavaServer Faces (JSF), альтернатива технологии JavaServer Pages (JSP). Фреймворк требует для функционирования валидные XML документы. Это означает, что веб-страницы должны быть созданы с использованием языка разметки XHTML. Facelets поддерживает все компоненты JSF и создаёт собственное дерево компонент, отражая view (см. MVC) для JSF-приложения. В Facelets полностью устранена широко известная проблема смешанного контента, присущая JavaServer Pages.

**Java Server Pages (JSP)** - это технология Java, которая позволяет веб-разработчикам и дизайнерам быстро разрабатывать и легко поддерживать насыщенные информацией динамические веб-страницы, которые используют существующие бизнес-системы. Технология JSP отделяет пользовательский интерфейс от создания контента, позволяя дизайнерам изменять общий макет страницы без изменения основного динамического контента.

**Facelets** - это первый язык объявления страниц без JSP, разработанный для JSF (Java Server Faces), который предоставил разработчикам JSF более простую и мощную модель программирования по сравнению с JSP. Он решает различные проблемы, возникающие в JSP для разработки веб-приложений.

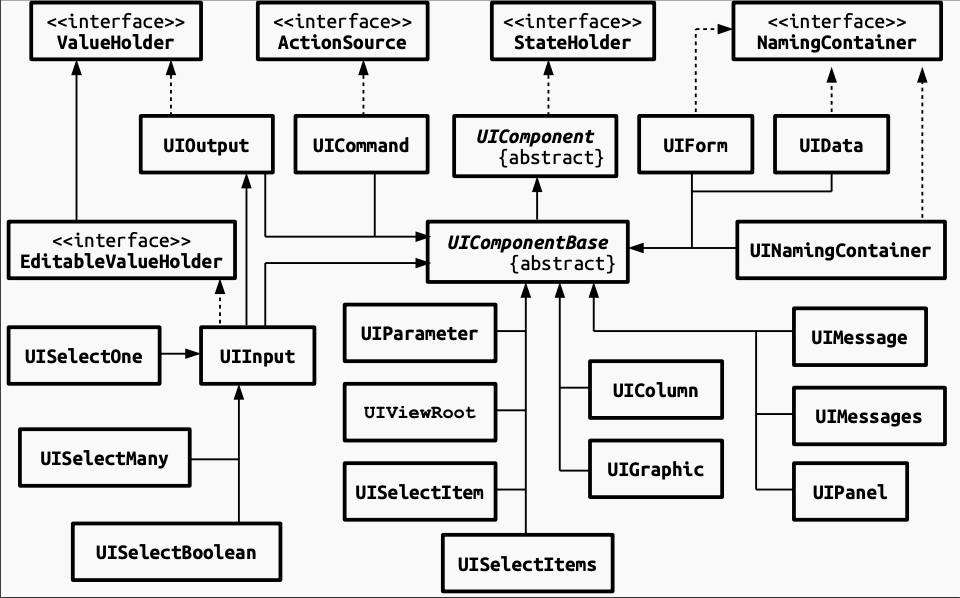
[](https://github.com/whoismuch/Laba3-WEB/blob/master/pictures/JspVsFacelets.png)

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

**Особенности реализации:**

* Интерфейс строится из компонентов.
* Компоненты расположены на страницах Facelets-шаблонах или 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.

**Иерархия классов**

[](https://github.com/whoismuch/Laba3-WEB/blob/master/pictures/classHierarchy.png)

-- 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**

JSF - это веб-среда MVC, управляемая запросами, для создания пользовательских интерфейсов с использованием компонентов.

И PrimeFaces/RichFaces/IceFaces являются компонентами/JSF-библиотеками, которые вы можете использовать поверх JSF

RichFaces: это библиотека компонентов с поддержкой Ajax с открытым исходным кодом для JavaServer Faces

ICEfaces: open-source, Java JSF расширение и богатые компоненты, Ajax без JavaScript

PrimeFaces: Структура Ajax с компонентами JSF

**Модель обработки событий в JSF-приложениях**

1. Отправляется request
2. Формирование представления
   * JSF Runtime формирует представление (начиная с UIViewRoot):
     + Создаются объекты компонентов.
     + Назначаются слушатели событий, конвертеры и валидаторы.
     + Все элементы представления помещаются в FacesContext.
   * Если это первый запрос пользователя к странице JSF, то формируется пустое представление.
   * Если это запрос к уже существующей странице, то JSF Runtime синхронизирует состояние компонентов представления с клиентом.
3. Получение значений компонентов
   * На стороне клиента все значения хранятся в строковом формате — нужна проверка их корректности:
     + Вызывается конвертер в соответствии с типом данных значения.
   * Если конвертация заканчивается успешно, значение сохраняется в локальной переменной компонента.
   * Если конвертация заканчивается неудачно, создаётся сообщение об ошибке, которое помещается в FacesContext.
4. Валидации значений компонентов
   * Вызываются валидаторы, зарегистрированные для компонентов представления.
   * Если значение компонента не проходит валидацию, формируется сообщение об ошибке, которое сохраняется в FacesContext.
5. Обновление значений компонентов
   * Если данные валидны, то значение компонента обновляется.
   * Новое значение присваивается полю объекта компонента.
6. Фаза вызова приложения
   * Управление передаётся слушателям событий.
   * Формируются новые значения компонентов.
7. JSF Runtime обновляет представление в соответствии с результатами обработки запроса.
   * Если это первый запрос к странице, то компоненты помещаются в иерархию представления.
   * Формируется ответ сервера на запрос.
   * На стороне клиента происходит обновление страницы.

Фазы жизненного цикла:

* Восстановление представления (Restore View) : для запрошенной страницы либо извлекается дерево компонентов, если страница уже открывалась, либо создается новое дерево компонентов, если страница запрашивается впервые. Для компонентов запрашиваемой страницы восстанавливаются их прошлые состояния (форма заполняет вводимыми значениями).
* Применение значений запроса (Apply Request Values) : анализ HTTP запроса, объектам дерева компонентов присваиваются соответствующие им значения из запроса. Если к компоненту подключен конвертер, то значение обрабатывается/конвертируется. При возникновении ошибки, все последующие шаги пропускаются. Если компонент ввода (UIInput) содержит валидатор и имеет свойство immediate="true", то этот компонент будет валидироваться в этой фазе. Также, при необходимости, события (нажатие кнопки или ссылки) добавляются в очередь событий.
* Выполнение проверок (Process Validations) : преобразование строковых значений в "локальные значения" и применение валидации дерева компонентов. Если в результате валидации компонента возникает ошибка, то она сохраняется и JSF пропускает все последующие шаги обработки запроса до фазы "Формирования ответа" для предупреждения пользователя об ошибке.
* Обновление значений модели (Update Model Values) : обновление свойства managed bean информацией из дерева компонентов.
* Выполнение приложения (Invoke Application) : JSF обрабатывает события, которые были сгенерированы нажатием на кнопки и ссылки. На данном этапе также решаются вопросы, связанные с навигацией приложения, если это необходимо. Если один из компонентов формы имеет свойство immediate="true", то он должен был быть обработан в фазе "Применение значений запроса".
* Формирование ответа (Render Response) : JSF создает ответ, основываясь на данных, полученных на предыдущих шагах. Информация страницы обновляется данными из managed bean и генерируется html страница с помощью Renderers. Если на предыдущих шагах происходили какие-либо ошибки, то они инкапсулируются в тег .
* Страница Java Server Faces представлена в виде дерева компонентов. Корневым компонентом дерева является UIViewRoot.

Первоначальный запрос - это запрос, который браузер выполняет для отображения страницы с ${customer.name}тегом. Обратная передача происходит, когда браузер публикует некоторые или все значения страницы, а затем та же самая страница, которая была отправлена ​​в первую очередь, возвращается клиенту. Это могло произойти, например, в результате ошибки проверки.

* initial request passes only Restore View & Render Response phases, while postback request process under all phases (Apply Request Values, Validations Phase, etc).
* Initial request is created by clicking a link, pasting an URL in address bar, while a postback request is create by posting a form by clicking a submit button or any post request.

Not all the time. All the 6 phases will be executed in case of an initial GET request with view parameters. – Shirgill Farhan Mar 21 '16 at 12:44 This answer is not 100% correct.

@ShirgillFarhanAnsari 's comment is correct and the answer is better updated. – Koray Tugay May 18 '16 at 7:51

* The initial and postback requests in JSF go through a JSF life cycle. When an initial request is processed, it only executes the Restore View and Render Response phases, because there is no user input or actions to process. On the other hand, when the life cycle handles a postback request, it executes all of the phases.

Moreover, JSF supports AJAX requests. An AJAX request consists of two parts: partial processing (the execute attribute) and partial rendering (the render attribute).

**Ajax Request Lifecycle**

An Ajax request varies from other typical JavaServer Faces requests, and its processing is also handled differently by the JavaServer Faces lifecycle.

As described in Partial Processing and Partial Rendering, when an Ajax request is received, the state associated with that request is captured by the javax.faces.context.PartialViewContext. This object provides access to information such as which components are targeted for processing/rendering. The processPartial method of PartialViewContext uses this information to perform partial component tree processing and rendering.

The execute attribute of the f:ajax tag identifies which segments of the server-side component tree should be processed. Because components can be uniquely identified in the JavaServer Faces component tree, it is easy to identify and process a single component, a few components, or a whole tree. This is made possible by the visitTree method of the UIComponent class. The identified components then run through the JavaServer Faces request lifecycle phases.

Similar to the execute attribute, the render attribute identifies which segments of the JavaServer Faces component tree need to be rendered during the render response phase.

During the render response phase, the render attribute is examined. The identified components are found and asked to render themselves and their children. The components are then packaged up and sent back to the client as a response.

Запрос Ajax отличается от других типичных запросов JavaServer Faces, и его обработка также по-разному обрабатывается жизненным циклом JavaServer Faces.

Как описано в разделах «Частичная обработка и частичная отрисовка» , при получении запроса Ajax состояние, связанное с этим запросом, фиксируется в javax.faces.context.PartialViewContext. Этот объект обеспечивает доступ к информации, например, о том, какие компоненты предназначены для обработки / рендеринга. processPartialМетод PartialViewContextиспользует эту информацию для выполнения частичного компонента обработки дерева и рендеринга.

executeАтрибут f:ajaxтег определяет , какие сегменты дерева компонентов на стороне серверов должны быть обработаны. Поскольку компоненты могут быть однозначно идентифицированы в дереве компонентов JavaServer Faces, легко идентифицировать и обрабатывать отдельный компонент, несколько компонентов или все дерево. Это стало возможным благодаря visitTreeметоду UIComponentкласса. Затем идентифицированные компоненты проходят этапы жизненного цикла запроса JavaServer Faces.

Подобно executeатрибуту, renderатрибут определяет, какие сегменты дерева компонентов JavaServer Faces должны быть визуализированы во время фазы ответа на визуализацию.

На этапе отклика на рендеринг renderпроверяется атрибут. Идентифицированные компоненты находят и просят отобразить самих себя и своих детей. Затем компоненты упаковываются и отправляются обратно клиенту в качестве ответа.

**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 представление 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.

process update render execute

execute = process and update = render

**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;

...

}

A JSF application is just like any other Java technology-based web application; it runs in a Java servlet container, and contains:

**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.**

The responsibilities of the application architect include the following:

* Registering back-end objects with the application so that all parts of the application have access to them
* Configuring backing beans and model beans so that they are instantiated with the proper values when a page makes reference to them
* Defining navigation rules for each of the pages in the application so that the application has a smooth page flow
* Packaging the application to include all the pages, objects, and other files so that the application can be deployed on any compliant container

web.xml is the deployment descriptor file and it is part of the servlet standard for web applications. It is used to determine how URLs map to servlets, which URLs require authentication, and other information. This file resides in the app's WAR under the WEB-INF/ directory.

faces-config.xml is usually the name for the application configuration resource file in JavaServer Faces technology and provides a portable configuration format (as an XML document) for configuring resources.An application architect creates one or more files, called application configuration resource files, that use this format to register and configure objects and to define navigation rules.

**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

Объявление управляемого объекта: имя объекта задается с помощью , полное имя класса - . Класс управляемого объекта обязан содержать конструктор без параметров.

> 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.

**Способ 1 — через faces-confg.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>Point</managed-bean-class>

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

</managed-bean>

</faces-config>

**Способ 2 (JSF 2.0) — с помощью аннотаций:**

@ManagedBean(name="customer")

@RequestScoped

public class CustomerBean {

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

private String areaCode;

}

**Класс FacesServlet:**

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

**8.Навигация в JSF-приложениях.**

<https://coderlessons.com/tutorials/veb-razrabotka/vyuchi-jsf/jsf-navigatsiia-po-stranitsam>

Правила навигации — это правила, предоставляемые JSF Framework, которые описывают, какое представление должно отображаться при нажатии кнопки или ссылки.

Правила навигации могут быть определены в файле конфигурации JSF с именемface-config.xml. Они могут быть определены в управляемых компонентах.

Правила навигации могут содержать условия, на основании которых может быть отображено полученное представление. JSF 2.0 также обеспечивает неявную навигацию, в которой нет необходимости определять правила навигации как таковые.

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:

* 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>

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

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

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

</h:outputLink>

**Форвард против перенаправления**

* JSF по умолчанию выполняет пересылку страницы сервера при переходе на другую страницу, и URL-адрес приложения не изменяется.
* Чтобы включить перенаправление страницы, добавьте Face-Redirect = True в конце имени представления.

<h:form>

<h3>Forward</h3>

<h:commandButton action = "page1" value = "Page1" />

<h3>Redirect</h3>

<h:commandButton action = "page1?faces-redirect = true" value = "Page1" />

</h:form>

In Java environments, client sessions are identified by a "jsessionid", a unique identifier that lets the server associate a series of client requests as being from the same client. Normally the jsessionid is held in a JSESSIONID cookie, but not all clients have cookies enabled. Servlet engines work around this limitation by adding the jsessionid into the URLs presented to the client so that each client click includes in the request URL itself the jsessionid value. Servlets provide a response.encodeURL() method that programmers are instructed to use when outputting URLs. This method gives the servlet environment the opportunity to add the jsessionid value to the URL when it deems it necessary.

You can get rid of jsessionid path fragment in URL by adding the following to web.xml:

<session-config>

<tracking-mode>COOKIE</tracking-mode>

</session-config>

It basically disables URL-rewriting.

**Реализуется экземплярами класса NavigationHandler.**

* Правила задаются в файле faces-confg.xml:

<navigation-rule>

<from-view-id>/pages/inputname.xhtml</from-viewid>

<navigation-case>

<from-outcome>sayHello</from-outcome>

<to-view-id>/pages/greeting.xhtml</to-view-id>

</navigation-case>

<navigation-case>

<to-view-id>/pages/goodbye.xhtml</to-view-id>

</navigation-case>

</navigation-rule>

-- Пример перенаправления на другую страницу:

<h:commandButton id="submit" action="sayHello" value="Submit" />

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

ODBC - это API данных, разработанный для языка программирования Java. Он был выпущен вместе с JDK 1.1 компанией Sun Microsystems (первоначальные владельцы Java). И его текущей версией является JDBC 4.0 (в настоящее время распространяется с JAVA SE6). Пакеты Java.sql и javax.sql содержат классы JDBC. Это интерфейс, который помогает клиенту получить доступ к системе баз данных, предоставляя методы для запроса и обновления данных в базах данных. JDBC больше подходит для объектно-ориентированных баз данных. Вы можете получить доступ к любой ODBC-совместимой базе данных, используя мост JDBC-ODBC.

**В чем разница между ODBC и JDBC?**

ODBC - это открытый интерфейс, который может использоваться любым приложением для связи с любой системой баз данных, а JDBC - это интерфейс, который может использоваться приложениями Java для доступа к базам данных. Следовательно, в отличие от JDBC, ODBC не зависит от языка. Но с помощью моста JDBC-к-ODBC Java-приложения могут также общаться с любой ODBC-совместимой базой данных.

Для взаимодействия Java-апплета с внешним сервером баз данных разработан специализированный протокол JDBC, который, фактически, сочетает функции шлюзования между интерпретатором мобильных Java-кодов и интерфейсом ODBC (Open Data Base Connectivity). JDBC - это разработанный JavaSoft прикладной программный SQL интерфейс API JDBC к базам данных. Этот API позволяет использовать стандартный набор процедур высокого уровня для доступа к различным БД.

Архитектура JDBC состоит из двух уровней: JDBC API, который обеспечивает связь между приложением и менеджером JDBC и драйвер JDBC API, который поддерживает связь между JDBC менеджером и драйвером (рис.3). Разработчики имеют возможность взаимодействовать напрямую с ODBC посредством моста JDBC-ODBC.

**Driver'ы**

**JDBC** – Java DataBase Connectivity – соединение с базами данных на Java. Это стандарт взаимодействия с БД. Его интерфейс поставляется в стандартной JDK в виде пакета java.sql. Какой бы библиотекой для работы с базой данных вы ни пользовались, в ее основе почти всегда лежит низкоуровневый JDBC.

**JDBC-драйвер** – реализация JDBC для определенной базы данных. В приложении может быть зарегистрировано несколько разных драйверов. При соединении к базе, нужный выбирается исходя из URL соединения. Например, для jdbc:mysql://localhost:3306/db\_name будет использован MySQL JDBC драйвер (при наличии). Драйверы делятся на 4 типа:

**Тип 1** – JDBC-ODBC мост. Делегирует работу с базой установленному в системе ODBC-драйверу. Платформозависимый. Не поддерживается с Java 8.

**Тип 2** – Native API. Делегирует работу с базой библиотеке, установленной в системе. Платформозависимый. Библиотеки в отличие от ODBC специфичны для конкретной базы, поэтому такой драйвер обычно работает быстрее.

**Тип 3** – драйвер сетевого протокола. Работает с промежуточным слоем (JavaEE сервером приложений), который транслирует запросы в сетевой протокол, с которым работает конкретная БД. Такой драйвер полностью реализован на Java.

**Тип 4** – драйвер протокола БД/тонкий драйвер. Полная реализация протокола взаимодействия с базой данных. Отличается от типа 3 тем, что здесь логика протокола находится не на внешнем промежуточном слое, а прямо в самом драйвере.

**Интерфейс Driver**

* java.sql.Driver
* Отвечает за связь с БД
* Метод Connection connect(String url, Properties info)
* Используется для написания драйверов для СУБД

Преза Гаврилова <https://se.ifmo.ru/documents/10180/1422934/prog-lec2.5v5.pdf/8f72469d-3693-9331-111c-054351cfa97e>

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

Достоинства

Среди достоинств ORM выделяют:

* наличие явного описания схемы БД, представленное в терминах какого-либо языка программирования, которое находится и редактируется в одном месте;
* возможность оперировать элементами языка программирования, т.е. классами, объектами, атрибутами, методами, а не элементами реляционной модели данных;
* возможность автоматического создания SQL-запросов, которая избавляет от необходимости использования языка для описания структуры БД (Data Definition Language) и языка манипулирования данными (Data Manipulation Language) при проектировании БД и изменении её схемы соответственно;
* не нужно создавать новые SQL-запросы при переносе на другую систему управления базами данных, поскольку за это отвечает низкоуровневый драйвер ORM.
* ORM избавляет от необходимости работы с SQL и проработки значительного количества программного кода, который зачастую однообразен и подвержен ошибкам.
* код, генерируемый ORM гипотетически проверен и оптимизирован, следовательно не нужно беспокоиться о его тестировании;
* развитые реализации ORM поддерживают отображение наследования и композиции на таблицы;
* ORM дает возможность изолировать код программы от подробностей хранения данных.

Недостатки

Среди недостатков ORM выделяются:

* Дополнительная нагрузка на программиста, которому, в случае использования ORM необходимо изучать этот некий «дополнительный слой» между программной и базой данных, который к тому же создает дополнительный уровень абстракции — объекты ORM. В связи с этим могут возникнуть вопросы соответствия особенностям ООП и соответствующим реляционным операциям. Эту проблему называют impedance mismatch, а сама реализация ORM ведет к увеличению объема программного кода и снижению скорости работы программы. Однако, с другой стороны, ORM наглядно и в одном месте концентрирует различие между реляционной и объектно-ориентированной парадигмами, что нельзя назвать недостатком;
* Появление трудно поддающихся отладке ошибок в программе, если присутствуют ошибки в реализации ORM, например, ошибки в реализации кэширования ORM, такие как согласование изменений в разных сессиях.
* Недостатки реализаций, которые могут иметь определенные ограничения и выдвигать определенные требования, например, требование собственной схемы базы данных и ограничение на средства создания базы данных. Также может отсутствовать возможность написать в явном виде SQL-запрос.
* Требуются отдельные таблицы в случае прямого отображения классов в таблицы и необходимости отображения атрибутов множественного характера.
* Если говорить о главном минусе ORM, снижении производительности, то причина этого состоит в том, что большинство из ORM нацелены на обработку значительного большего количества различных сценариев использования данных, чем в случае отдельного приложения. В случае небольших проектов, которые не сталкиваются с высокой нагрузкой, применение ORM очевидно, особенно, если учесть такой важный критерий разработки, как время. «То, что с легкостью пишется с использованием ORM за неделю, можно реализовывать ни один месяц собственными усилиями».[Источник 2]

ORM ― Object/Relational Mapping ― преобразование данных из объектной формы в реляционную и наоборот.

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

Но время шло и разработчики каждый раз сталкивались с необходимостью писать однотипный и ненужный "обслуживающий" код (так называемый Boilerplate code) для тривиальных операций по сохранению Java объектов в БД и наоборот, созданию Java объектов по данным из БД. И тогда для решения этих проблем на свет появилось такое понятие, как ORM.

ORM — Object-Relational Mapping или в переводе на русский объектно-реляционное отображение. Это технология программирования, которая связывает базы данных с концепциями объектно-ориентированных языков программирования. Если упростить, то ORM это связь Java объектов и записей в БД:

EclipseLink is more standards compliant, since it is the reference implementation for JPA 2, Hibernate has some compliancy issues, but is more mature.

One of the main benefits of EclipseLink is that you can call native SQL functions directly in your JPQL queries. In Hibernate this is not directly possible.

But Hibernate has a bigger community, better documentation and also better error messages.

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

Одним из основных преимуществ EclipseLink является то, что вы можете вызывать собственные функции SQL непосредственно в своих запросах JPQL. В Hibernate это невозможно напрямую.

Но Hibernate имеет большее сообщество, лучшую документацию, а также лучшие сообщения об ошибках.

**Hibernate:**

ORM-фреймворк от Red Hat, разрабатывается с 2001 г.

Ключевые особенности:

* Таблицы БД описываются в XML-файле, либо с помощью аннотаций.
* 2 способа написания запросов — HQL и Criteria API.
* Есть возможность написания native SQL запросов.
* Есть возможность интеграции с Apache Lucene для полнотекстового поиска по БД (Hibernate Search).

**EclipseLink**

ORM-фреймворк от Eclipse Foundation.

Ключевые особенности:

* Основан на кодовой базе Oracle TopLink.
* Является эталонной реализацией (reference implementation) для JPA.

Hibernate одна из самых популярных открытых реализаций последней версии спецификации (JPA 2.1). Даже скорее самая популярная, почти стандарт де-факто. То есть JPA только описывает правила и API, а Hibernate реализует эти описания, впрочем у Hibernate (как и у многих других реализаций JPA) есть дополнительные возможности, не описанные в JPA (и не переносимые на другие реализации JPA

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

ORM — это по сути концепция о том, что Java объект можно представить как данные в БД (и наоборот). Она нашла воплощение в виде спецификации JPA — Java Persistence API.

Спецификация — это уже описание Java API, которое выражает эту концепцию. Спецификация рассказывает, какими средствами мы должны быть обеспечены (т.е. через какие интерфейсы мы сможем работать), чтобы работать по концепции ORM. И как использовать эти средства.

Реализацию средств спецификация не описывает. Это даёт возможность использовать для одной спецификации разные реализации. Можно упростить и сказать, что спецификация — это описание API.

Реализации JPA ещё называют JPA Provider. Одной из самых заметных реализаций JPA является Hibernate. Поэтому, предлагаю её и рассмотреть.

Java-стандарт (JSR 220, JSR 317), который определяет:

* как Java-объекты хранятся в базе;
* API для работы с хранимыми Java-объектами;
* язык запросов (JPQL);
* возможности использования в различных окружениях

###Что даёт использование JPA?

* Достижение лучшей переносимости.
* Упрощение кода.
* Сокращение времени разработки.
* Независимость от ORM-фреймворков.

**Допольнительные вопросы к себе (дада, у меня шиза)**

1. Атрибут xmlns в .html
2. Подпредставления в jsf
3. f:facet
4. Зачем??? Правила навигации между страницами приложения должны быть заданы в отдельном конфигурационном файле.
5. Render и что делет faces servlet

**Вопрос1 (очень вонючий) action in managed bean returns null, empty string or void. Any differents ?**

There is actually a technical difference with regard to null and non-null outcomes. The null (or void) outcome will reuse the same view. A non-null outcome will trash the current view and create a brand new view, even if it returns the same view identifier. The "view" is here the UIViewRoot instance including all of its state. The empty string represents "current view ID".

Notably @ViewScoped beans are affected by this. They will be trashed and recreated along with the view. If you're however using a @RequestScoped bean, then you indeed won't notice any technical difference as to bean behavior. It won't be trashed when the view gets trashed, but only when the request ends. In other words, even when the same view is recreated within the same request, the same @RequestScoped bean will be used on it.

The right approach depends on the concrete functional requirements. If you want to return to the same view, just return null or void (not empty string). It's generally unnecessary to create a new view in such case. Only if you're having view build time tags (JSTL and friends) in current view whose model values have been changed in bean action method, then you'd need to force a rebuild of the view by returning an empty string. If you want to navigate to a different view, return the desired view identifier, preferably along with faces-redirect=true, for sure if it's idempotent (bookmarkable).

Yes both mean same .But for Action methods we generally prefer using pagereference methods so that later it provides us the flexibility to provide navigation.While returning void will not flush the view state as well but generally later we have no flexibility to make navigation through this method – Mohith Shrivastava Jul 17 '13 at 19:01

You should return void if there is no possibility of a redirect. This makes it clear to other developers that may later view the source code that there will be no redirect from this action function without examining the entire function. This is useful when the function spans hundreds of lines and includes complex control structures. In the application I am working on finishing up, the prior developer always returned a null page reference instead of using void, and it significantly slowed down my progress.

**Something about @ViewScoped and @RequestScoped**

A @ViewScoped bean lives exactly as long as a JSF view. It usually starts with a fresh new GET request, or with a navigation action, and will then live as long as the enduser submits any POST form in the view to an action method which returns null or void (and thus navigates back to the same view). Once you refresh the page, or return a non-null string (even an empty string!) navigation outcome, then the view scope will end.

A @RequestScoped bean lives exactly as long a HTTP request. It will thus be garbaged by end of every request and recreated on every new request, hereby losing all changed properties.

A @ViewScoped bean is thus particularly more useful in rich Ajax-enabled views which needs to remember the (changed) view state across Ajax requests. A @RequestScoped one would be recreated on every Ajax request and thus fail to remember all changed view state. Note that a @ViewScoped bean does not share any data among different browser tabs/windows in the same session like as a @SessionScoped bean. Every view has its own unique @ViewScoped bean.

**Вопрос2 (не менее вонючий) fetch.TYPE.LAZY and fetch.TYPE.EAGER**

LAZY = fetch when needed EAGER = fetch immediately

Sometimes you have two entities and there's a relationship between them. For example, you might have an entity called University and another entity called Student and a University might have many Students:

The University entity might have some basic properties such as id, name, address, etc. as well as a collection property called students that returns the list of students for a given university:

A university has many students

public class University {

private String id;

private String name;

private String address;

private List<Student> students;

// setters and getters

}

Now when you load a University from the database, JPA loads its id, name, and address fields for you. But you have two options for how students should be loaded:

1. To load it together with the rest of the fields (i.e. eagerly), or
2. To load it on-demand (i.e. lazily) when you call the university's getStudents() method.

When a university has many students it is not efficient to load all of its students together with it, especially when they are not needed and in suchlike cases you can declare that you want students to be loaded when they are actually needed. This is called lazy loading.

Here's an example, where students is explicitly marked to be loaded eagerly:

@Entity

public class University {

@Id

private String id;

private String name;

private String address;

@OneToMany(fetch = FetchType.EAGER)

private List<Student> students;

// etc.

}

And here's an example where students is explicitly marked to be loaded lazily:

@Entity

public class University {

@Id

private String id;

private String name;

private String address;

@OneToMany(fetch = FetchType.LAZY)

private List<Student> students;

// etc.

}

* Чтобы ленивая загрузка работала, сеанс JDBC должен быть открыт, когда целевые сущности хотят загрузить в память, вызывая метод получения (например getStudents()), но иногда это невозможно, потому что к тому времени этот метод вызывается, сеанс уже закрыт и объект отсоединен. Точно так же иногда у нас есть архитектура клиент / сервер (например, клиент Swing / сервер JEE), и объекты / DTO передаются по сети клиенту, и опять же, чаще всего в этих сценариях ленивая загрузка не работает из-за того, как объекты сериализуются по сети. -
* Чтобы сэкономить память, ленивая загрузка обычно используется для отношений один ко многим и многие ко многим. Один к одному, как правило, используется Eager
* Используйте ленивый везде и в JQL, быстро получайте, когда вам нужно. Зачем? Что ж, я продолжаю сталкиваться с системой, которая загружает 6 соединений, потому что все ToOne хотят. Фактически, я хочу вернуть старую ленивую спящую среду по умолчанию. Вы никогда не знаете свои варианты использования заранее, и одному понадобится ленивый, а другой нетерпеливый ... вместо этого используйте ленивый везде, а когда вам нужно нетерпение, создайте для него запрос
* «Использование или неиспользование выборки EAGER может привести к проблемам с производительностью». В том конкретном случае, когда лениво инициализированное поле является дорогостоящим для доступа и используется нечасто, ленивая выборка улучшит производительность. Но в случае, когда переменная часто используется, отложенная инициализация может фактически снизить производительность , требуя большего количества обращений к базе данных, чем активная инициализация. Я бы посоветовал правильно, а не догматично применять FetchType
* Тип Lazy выборки по умолчанию выбирается Hibernate, если вы явно не отметили Eager тип выборки. Чтобы быть более точным и кратким, разница может быть указана ниже.

**When to use eager loading:**

* In "one side" of one-to-many relations that you sure are used every where with main entity. like User property of an Article. Category property of a Product.
* Обычно, когда отношений не слишком много, и активная загрузка будет хорошей практикой, чтобы уменьшить количество запросов на сервер.

**When to use lazy loading**

* Almost on every "collection side" of one-to-many relations. like Articles of User or Products of a Category
* You exactly know that you will not need a property instantly.