Архитектура веб-приложений. Шаблон MVC. Model 1 и Model 2.

В компьютерных технологиях **трёхуровневая архитектура**, синоним **трёхзвенная архитектура** предполагает наличие следующих компонентов приложения: клиентское приложение (обычно говорят «тонкий клиент» или терминал), подключенное к серверу приложений, который в свою очередь подключен к серверу базы данных.

Терминал — это интерфейсный (обычно графический) компонент, который представляет первый уровень, собственно приложение для конечного пользователя. Первый уровень не должен иметь прямых связей с базой данных (по требованиям безопасности), быть нагруженным основной бизнес-логикой (по требованияммасштабируемости) и хранить состояние приложения (по требованиям надежности). На первый уровень может быть вынесена и обычно выносится простейшая бизнес-логика:

Сервер приложений располагается на втором уровне. На втором уровне сосредоточена бóльшая часть бизнес-логики. Вне его остаются фрагменты, экспортируемые на терминалы (см.выше), а также погруженные в третий уровень хранимые процедуры и триггеры.

Сервер базы данных обеспечивает хранение данных и выносится на третий уровень. Обычно это стандартнаяреляционная или объектно-ориентированная СУБД.

Достоинства

масштабируемость

конфигурируемость

высокая безопасность

высокая надёжность

низкие требования к скорости канала (сети) между терминалами и сервером приложений

низкие требования к производительности и техническим характеристикам терминалов

Недостатки

Недостатки вытекают из достоинств. По сравнению с клиент-серверной или файл-серверной архитектурой можно выделить следующие недостатки трёхуровневой архитектуры:

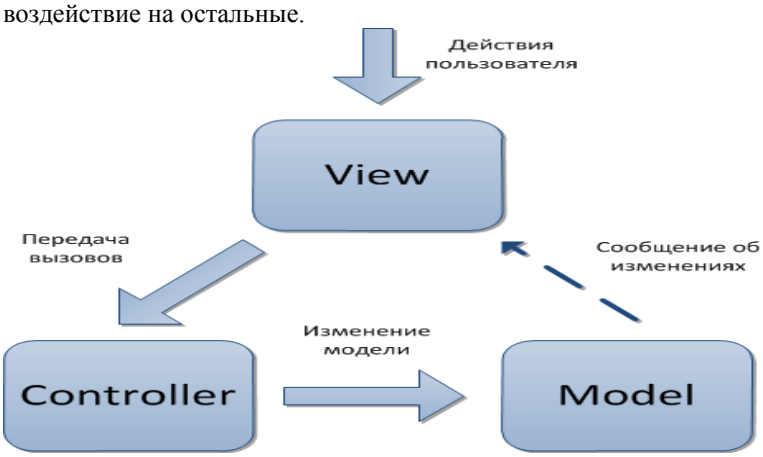
более высокая сложность создания приложений;

сложнее в разворачивании и администрировании;

высокие требования к производительности серверов приложений и сервера базы данных, а, значит, и высокая стоимость серверного оборудования;

высокие требования к скорости канала (сети) между сервером базы данных и серверами приложений.

Model-view-controller (**MVC**, «Модель-представление-поведение», «Модель-представление-контроллер») —архитектура программного обеспечения, в которой модель данных приложения, пользовательский интерфейс и управляющая логика разделены на три отдельных компонента так, что модификация одного из компонентов оказывает минимальное воздействие на остальные.



MVC состоит из трех компонент: View (представление, пользовательский интерфейс), Model (модель, ваша бизнес логика) и Controller (контроллер, содержит логику на изменение модели при определенных действиях пользователя, реализует Use Case). Основная идея этого паттерна в том, что и контроллер и представление зависят от модели, но модель никак не зависит от этих двух компонент. Это как раз и позволяет разрабатывать и тестировать модель, ничего не зная о представлении и контроллерах. В идеале контроллер так же ничего не должен знать о представлении (хотя на практике это не всегда так), и в идеале для одного представления можно переключать контроллеры, а так же один и тот же контроллер можно использовать для разных представлений (так, например, контроллер может зависеть от пользователя, который вошел в систему). Пользователь видит представление, на нем же производит какие-то действия, эти действия представление перенаправляет контроллеру и подписывается на изменение данных модели, контроллер в свою очередь производит определенные действия над моделью данных, представление получает последнее состояние модели и отображает ее пользователю.

Model 1 и Model 2.

В ранних спецификациях JSP были выделены два подхода к формированию приложений, использующих JSP . Эти подходы, названные Моделью 1 и Моделью 2 (JSP Model 1/2 architectures), отличаются, по существу, тем, какой компонент выполняет большую часть обработки запроса. В Модели 1 (рис. 1), JSP страница сама обрабатывает входящий запрос и генерирует ответ для клиента. Разделение представления и содержания присутствует, так как весь доступ к данным выполняется через компоненты Java-beans. Хотя, вероятно, Модель 1 вполне подходит для простых приложений, ее использование в сложных системах нежелательно. Неразборчивое использование этой архитектуры обычно ведет к большому количеству скриптлетов или Java-кода, внедренного в JSP-страницу, особенно, если необходимо выполнить объемную обработку запроса. Хотя это и не проблема для Java-разработчиков , это становится проблемой, если ваши JSP

страницы создаются и поддерживаются дизайнерами, что является обычной практикой при разработке крупных проектов. В конечном счете, это может вести к неясному распределению ролей и обязанностей между членами команды, вызывая головные боли у руководителей проекта, которых можно было бы легко избежать.
Предназначена для проектирования приложений небольшого масштаба и сложности. За обработку данных и представления отвечает один и тот же компонент (сервлет или JSP).

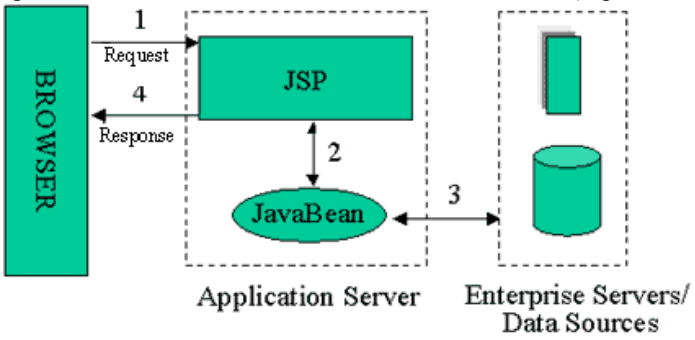


Рисунок 1. Архитектура JSP Модель 1.
Архитектура Модели 2 (рис.2), является гибридной технологией обработки динамического содержания, так как она комбинирует использование сервлетов и JSP. Это позволяет воспользоваться преимуществами обеих технологий, применяя JSP для реализации уровня представления и сервлеты для объемной обработки данных. В этой схеме сервлет действует как *контроллер* и отвечает за обработку запроса и создание компонентов Java-beans, используемых JSP, а также в зависимости от действий пользователя принимает решение, какой JSP-странице перенаправить запрос. Обратите внимание, что непосредственно в JSP-странице нет никакой логики обработки информации; она просто отвечает за извлечение любых объектов или beans, которые, возможно, были предварительно созданы сервлетом, и получение динамического содержания от этого сервлета для включения в статические шаблоны. По-моему, этот подход в общем случае приводит к наиболее полному отделению представления от содержания, и позволяет упорядочить роли и обязанности программистов и дизайнеров страниц в вашей команде разработчиков. Фактически, чем более сложным является ваше приложение, тем больше будет выгода от использования Модели 2.
Предназначена для проектирования достаточно сложных веб-приложений. За обработку и представление данных отвечают разные компоненты (сервлеты и JSP).

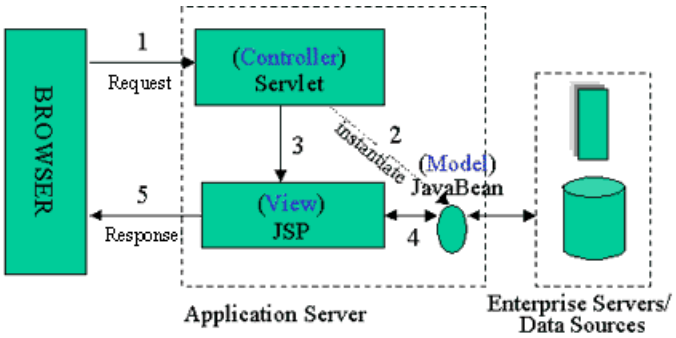


Рисунок 2. Архитектура JSP Модели 2.

Технология JavaServer Faces. Особенности, отличия от сервлетов и JSP, преимущества и недостатки. Структура JSF-приложения. Использование JSP-страниц в JSF-приложениях.

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.
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.
Преимущества и недостатки JSF
Четкое разделение бизнес-логики и интерфейса
Управление сохраняемостью 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)
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
Использование JSP-страниц 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 JSP-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.
Например, при изменении неких атрибутов на JSP-странице, а затем при обновлении ее, можно заметить что ничего не изменилось. (т.е. теги на странице обращаются к текущему состоянию компонента). Состояние компонент может быть модифицировано контроллером.

JSF-компоненты - особенности реализации, иерархия классов. Модель обработки событий в JSF-приложениях.

Особенности реализации:

- 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.

```
<f:view>
<h:form>
...
<f:subview>
<h:inputText id="txtMail" value="#{GuestBean.mail}" style="width:100%" />
</f:subview>
... другие faces-теги...
</h:form>
</f:view>
```

Некоторые компоненты JSF:

`<f:subview>` - «подпредставление»
Тег <f:subview> можно использовать на родительской либо на вложенной странице, но не на обеих одновременно. В JSF 1.2 <f:subview>, в отличие от предыдущих версий, не является обязательным элементом.
Листинг 6. Подпредставления на странице contacts.jsp

```
 <body>
<f:view>
  <h3>Contacts (2nd version)</h3>
  <h:messages infoClass="infoClass" errorClass="errorClass"
    layout="table" globalOnly="true" />
  <h:form>
    <h:commandLink binding="#{contactController.addNewCommand}"
      action="#{contactController.addNew}" value="Add New..." />
  </h:form>
  <f:subview id="form">
    <jsp:include page="form.jsp" />
  </f:subview>
  <f:subview id="listing">
    <jsp:include page="listing.jsp" />
  </f:subview>
</f:view>
</body>
```

<h:selectOneMenu> - меню с возможностью выбора одного элемента из списка

Листинг 7. Использование <h:selectOneMenu> для выбора группы

```
<%-- Group --%>
<h:outputLabel value="Group" for="group" accesskey="g" />
<h:selectOneMenu id="group" validatorMessage="required"
  value="#{contactController.selectedGroupId}">
  <f:selectItems value="#{contactController.groups}" />
  <f:validateLongRange minimum="1" />
</h:selectOneMenu>
<h:message for="group" errorClass="errorClass" />
```

При отправке формы на сервер устанавливается значение свойства selectedGroupId. Оно используется в методepersist(), который связан с кнопками добавления и редактирования контактов, для поиска нужной группы в репозитории.

<h:selectOneRadio> - группа переключателей

Благодаря конвертеру для перечислений значения в форме могут быть напрямую связаны со свойством type контакта (см. листинг 15):

Листинг 15. Привязка компонента непосредственно к свойству contact.type

```
<h:selectOneRadio id="type" value="#{contactController.contact.type}">
  <f:selectItem itemValue="PERSONAL" itemLabel="personal" />
  <f:selectItem itemValue="BUSINESS" itemLabel="business" />
</h:selectOneRadio>
```

В листинге 15 также показан пример использования компонента <f:selectItem> для создания отдельных элементов. Встроенный конвертер перечислений ожидает на входе строковые значения — названия элементов, составляющих перечисление. Вместо <h:selectOneRadio> можно так же использовать <h:selectOneListbox>, как в листинге 16:

Листинг 16. Использование компонента <h:selectOneListbox>

```
  <h:selectOneListbox id="type" value="#{contactController.contact.type}">
  <f:selectItem itemValue="PERSONAL" itemLabel="personal"/>
  <f:selectItem itemValue="BUSINESS" itemLabel="business"/>
</h:selectOneListbox>
```

<h:selectManyCheckbox> - группа флажков с многочисленной возможностью выбора

Кроме этого у класса Contact также существует отношение типа “многие-ко-многим” с классом Tag через свойство tags. Для работы с таким типом отношений используется компонент <h:selectManyCheckbox>

Листинг 10. Использование selectManyCheckbox вместе со свойством Contact.tags

```
<h:selectManyCheckbox id="tags"
  value="#{contactController.selectedTagIds}">
  <f:selectItems value="#{contactController.availableTags}" />
</h:selectManyCheckbox>
```

Главным отличием <h:selectManyCheckbox> является то, что он связан с массивом элементов типа long, а не одним long, Класс UISelectMany является предком не только компонента <h:selectManyCheckbox>, но и <h:selectManyListbox>вместе с <h:selectManyMenu>.

Листинг 12. <selectManyListbox>

```
<h:selectManyListbox id="tags" value="#{contactController.selectedTagIds}">
  <f:selectItems value="#{contactController.availableTags}"/>
</h:selectManyListbox>
```

Листинг 13. <selectManyMenu>

```
<h:selectManyMenu id="tags" value="#{contactController.selectedTagIds}">
  <f:selectItems value="#{contactController.availableTags}"/>
</h:selectManyMenu>
```

<h:inputTextarea> - поле ввода

Листинг 17. Использование компонентов <h:selectBooleanCheckbox> и <h:textArea>

```
<h:inputHidden value="#{contactController.contact.id}" />
...
<%-- active --%>
<h:outputLabel value="Active" for="active" accesskey="a" />
<h:selectBooleanCheckbox id="active"
  value="#{contactController.contact.active}" />
<h:message for="active" errorClass="errorClass" />
<%-- Description --%>
...
```

```
<h:outputLabel value="Description" for="description"
  accesskey="d" style="font: large;" />
<h:inputTextarea id="description" cols="80" rows="5"
  value="#{contactController.contact.description}" />
<h:message for="description" errorClass="errorClass" />
```

Элемент <h:inputTextarea> содержит два дополнительных атрибута, задающих размер области: cols="80" rows="5". Связывание компонентов со свойствами объектов работает как прежде.

Иерархия классов (фрагмент)

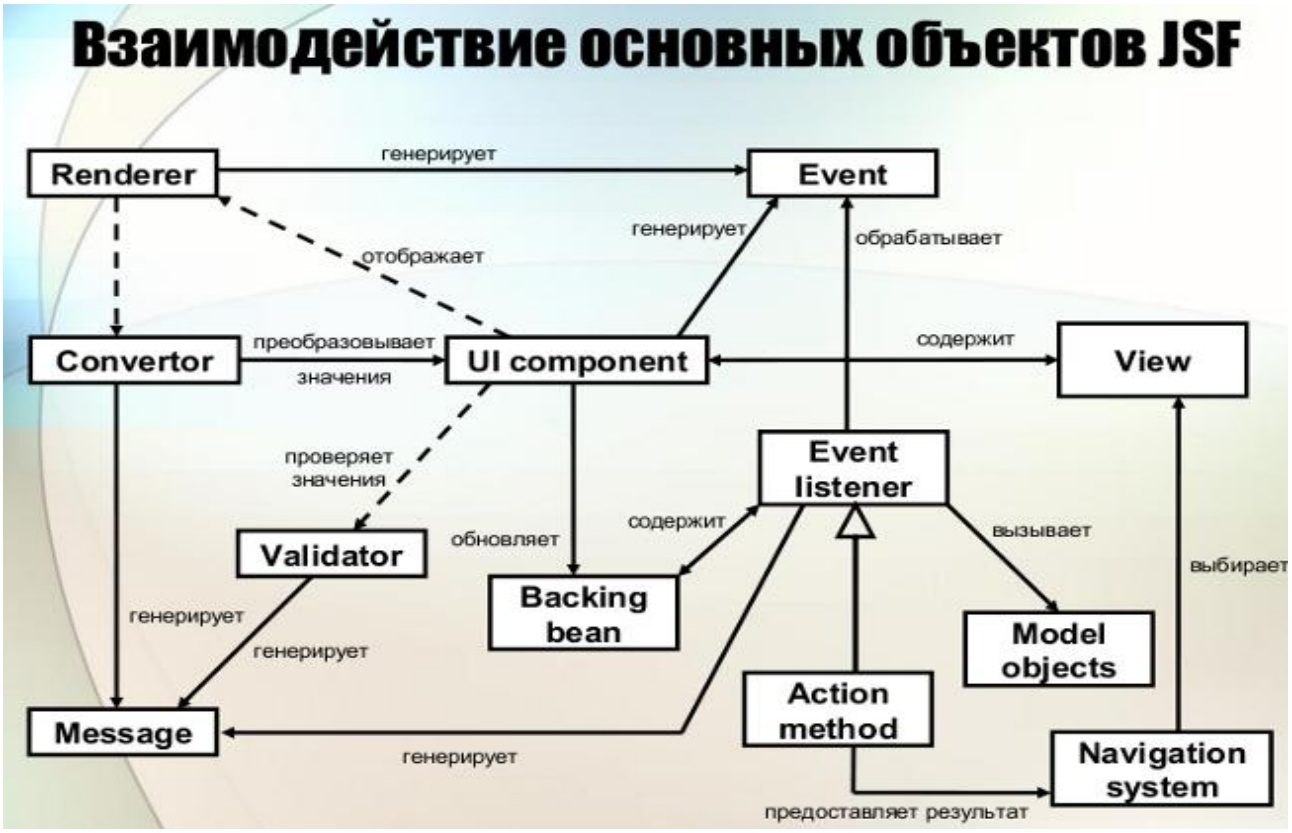
1. javax.faces.component.UIComponent
   1.1. javax.faces.component.UIComponentBase
      1.1.1. javax.faces.component.UIOutput
         1.1.1.1. javax.faces.component.UIInput
            1.1.1.1.1. javax.faces.component.UISelectOne
            1.1.1.1.2. javax.faces.component.UISelectMany

. . .

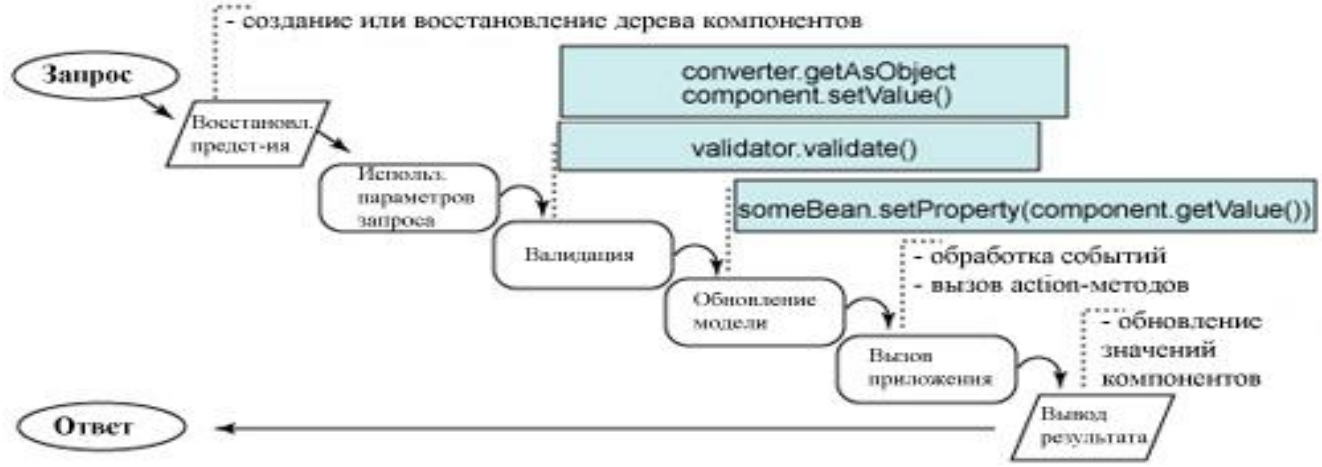
Модель обработки событий

Жизненный цикл обработки запроса в приложениях JSF состоит из следующих фаз:
Восстановление представления
Использование параметров запроса; обработка событий
Проверка данных; обработка событий
Обновление данных модели; обработка событий
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.



Шесть фаз цикла обработки запроса в JSF



Конвертеры и валидаторы данных.

Конвертеры данных в JSF

Конвертация – это процесс преобразования данных к нужным типам. В процессе конвертации строковые поля форм преобразуются в даты (класс Date), примитивные типы float, в объекты типа Float и т.д. JSF позволяет использовать как встроенные, так и специально созданные для данного приложения конвертеры. Вначале мы расскажем об использовании стандартных конвертеров JSF, а затем подробно рассмотрим создание специализированных конвертеров.

Стандартные конвертеры JSF

В стандартную поставку JSF входит множество стандартных конвертеров данных, благодаря чему большая часть конвертации происходит автоматически. В таблице 1 приведены идентификаторы стандартных конвертеров и реализующие их классы. Они используются в JSF для преобразования строк к простым типам.

Стандартные конвертеры

Конвертер	Реализующий класс
javax.faces.BigDecimal	javax.faces.convert.BigDecimalConverter
javax.faces.BigInteger	javax.faces.convert.BigIntegerConverter
javax.faces.Boolean	javax.faces.convert.BooleanConverter
javax.faces.Byte	javax.faces.convert.ByteConverter
javax.faces.Character	javax.faces.convert.CharacterConverter
javax.faces.DateTime	javax.faces.convert.DateTimeConverter
javax.faces.Double	javax.faces.convert.DoubleConverter
javax.faces.Float	javax.faces.convert.FloatConverter

Таким образом, если поле формы связано со свойством типа int или Integer, то конвертация происходит автоматически. В листинге 19 показан компонент, использующийся в нашем демонстрационном приложении для управления контактами. Он связан непосредственно со свойством age с помощью выражения #{contactController.contact.age}:

Связывание со свойством age: конвертация выполняется автоматически

```
<%-- age --%>
<h:outputLabel value="Age" for="age" accesskey="age" />
<h:inputText id="age" size="3" value="#{contactController.contact.age}">
</h:inputText>
```

Указание формата даты

```
<%-- birthDate --%>
<h:outputLabel value="Birth Date" for="birthDate" accesskey="b" />
  <h:inputText id="birthDate" value="#{contactController.contact.birthDate}">
    <f:convertDateTime pattern="MM/yyyy"/>
  </h:inputText>
<h:message for="birthDate" errorClass="errorClass" />
```

Специализированные конвертеры JSF

Специализированные конвертеры необходимы, если приходится преобразовывать значения полей в объекты типов, специфичных для данного приложения, например:
String в объект типа PhoneNumber (поля PhoneNumber.areaCode, PhoneNumber.prefix, ...)
String в объект типа Name (поля Name.first, Name.last)
String в объект типа ProductCode (поля ProductCode.partNum,ProductCode.rev, ...)
String в объект типа Group
String в объект типа Tags
Для создания специализированного конвертера необходимо следующее:
Создать класс, реализующий интерфейс Converter (полное имя javax.faxes.convert.Converter).
Реализовать метод getAsObject(), который будет вызываться для преобразования строкового значения поля в объект (например, типа PhoneNumber).
Реализовать метод getAsString, который будет вызываться для получения строкового представления объекта (например, типа PhoneNumber).
Зарегистрировать конвертер в контексте Faces. После того, как конвертеры созданы, необходимо указать JSF, что их необходимо использовать каждый раз, когда значения связываются со свойствами типа Group или Tag. Для этого необходимо зарегистрировать конвертеры в файле faces-config.xml, используя элемент <converter>.

Регистрация конвертеров в файле 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 в JSF

Главной целью конвертации и валидации является подготовка данных для обновления объектов модели. Таким образом, к моменту вызова методов, реализующих логику приложения, можно сделать определенные выводы о состоянии модели. Конвертация и валидация позволяют сконцентрироваться на бизнес-логике приложения, а не на утомительных проверках ввода, таких как проверка на null, на длину, на границы массивов и т.д.
В JSF существует четыре варианта того, как может происходить валидация:
С помощью встроенных компонентов
На уровне приложения
С помощью проверочных методов серверных объектов (inline-валидация)
С помощью специализированных компонентов, реализующих интерфейсValidator
Ниже мы рассмотрим все эти варианты.

Стандартная валидация

JSF включает в себя три стандартных компонента для валидации:
DoubleRangeValidator: Проверяет, что значение компонента укладывается в интервал, определяемый нижней границей, верхней границей или и тем, и другим. Значение должно быть числом.
LongRangeValidator: Проверяет, что значение укладывается в интервал, определяемый нижней границей, верхней границей или и тем, и другим. Значение должно быть числом, преобразуемым к типу long.
LengthValidator: Проверяет, что длина значения укладывается в интервал, определяемый нижней границей, верхней границей или и тем, и другим. Значение должно быть типа String.
В нашем примере возраст в данных контакта может быть любым числом. Но поскольку такие значения, как –2, не имеют смысла, стоит связать это поле с правилом валидации.

Валидация возрастных данных с помощью элемента <f:validateLongRange>

```
<%-- 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" />
```

Проверка длины строки свойства firstName

```
<%-- Имя (first name) --%>
<h:outputLabel value="First Name" for="firstName" accesskey="f" />
<h:inputText id="firstName" label="First Name" required="true"
 value="#{contactController.contact.firstName}" size="10" >
  <f:validateLength minimum="2" maximum="25" />
</h:inputText>
<h:message for="firstName" errorClass="errorClass" />
```

Валидация уровня приложения

Под валидацией уровня приложения понимается непосредственно бизнес-логика. В JSF она отделена от первичной валидации форм и их полей. Как правило, валидация уровня приложения заключается в добавлении в методы управляемых bean-объектов кода, который использует модель приложения для проверки уже помещенных в нее данных. Допустим, пользователь нажал на кнопку, связанную с action-методом, исполняемом на этапе вызова приложения. Перед тем, как с данными будут произведены какие либо действия (что, как правило, происходит в фазе обновления модели), можно проверить, являются ли введенные данные корректными с точки зрения бизнес-правил приложения.

Валидация уровня приложения в классе контроллера

```
public class ContactController {
  public String persist() {
     /* Валидация уровня приложения. */
      try {
         contact.validate();
      } catch (ContactValidationException contactValidationException) {
         addErrorMessage(contactValidationException.getLocalizedMessage());
         return null;}
    /* Форма доступна, ссылка для добавления - нет. */
    form.setRendered(false);
    addNewCommand.setRendered(true);
    /* Добавление сообщения о статусе операцмм. */
    if (contactRepository.persist(contact) == null) {
       addStatusMessage("Added " + contact);
    } else {
       addStatusMessage("Updated " + contact);}
    return "contactPersisted";}
  private void addErrorMessage(String message) {
     FacesContext.getCurrentInstance().addMessage(null, new FacesMessage(
          FacesMessage.SEVERITY_ERROR, message, null));}
```

Как видно из листинга 31, метод persist() вызывает метод validate() класса Contact. Он обрабатывает все исключения и преобразует их в сообщения об ошибках – объекты типа FacesMessage. В случае выдачи исключения onnull, что означает: *оставаться на текущем представлении и не переходить к следующему*.

Валидация в слое модели, а не контроллера

```
public class Contact implements Serializable {
   public void validate() throws ContactValidationException {
     if ((homePhoneNumber == null || ''.equals(homePhoneNumber))  &
        (workPhoneNumber == null || ''.equals(workPhoneNumber))  &
        (mobilePhoneNumber == null || ''.equals(mobilePhoneNumber))) {
       throw new ContactValidationException("At least one phone number" + "must be set");    }}
```

Валидация уровня приложения проста и легко реализуема. Ее преимущества:
простота реализации;
отсутствие необходимости специального класса (специализированного валидатора);
отсутствие необходимости указывать валидатор при разработке страниц представления;
Недостатком валидации уровня приложения является то, что она выполняется после других форм валидации (стандартной, специализированной и компонентной), поэтому сообщения об ошибках появляются только после того, как вся остальная валидация прошла успешно.
В целом валидацию уровня приложения следует использовать только там, где необходима проверка с учетом бизнес-логики.

Специализированная валидация в серверных объектах JavaBean

Для типов данных, не поддерживаемых стандартными валидаторами JSF, например, адресов электронной почты, почтовых индексов и т.д., приходится создавать собственные валидирующие компоненты. Кроме этого

специализированные валидаторы полезны там, где требуется полный контроль над сообщениями, выводимыми пользователю. JSF позволяет создавать встраиваемые валидирующие компоненты, которые можно использовать в различных Web-приложениях.
В качестве альтернативы созданию отдельного валидатора можно помещать специализированную валидацию в методы серверных объектов JavaBean.

Автономные специализированные валидаторы

JSF позволяет создавать подключаемые валидирующие компоненты, которые можно использовать в различных Web-приложениях.
Для создания валидатора необходимо сделать следующее:
класс, реализующий интерфейс Validator (javax.faces.validator.Validator).
Реализовать метод validate().
Зарегистрировать валидатор в файле faces-config.xml.
Использовать тег <f:validator/> на страницах JSP.
Далее мы остановимся на каждом из этих шагов, рассмотрев пример создания специализированного валидатора.

Регистрация валидатора в файле faces-config.xml

```
<validator>
  <validator-id>arcmind.zipCode</validator-id>
  <validator-class>com.arcmind.validators.ZipCodeValidator</validator-class>
</validator>
```

Использование тега <f:validator> на страницах JSP

```
  <%-- почтовый индекс (zip) --%>
<h:outputLabel value="Zip" for="zip" accesskey="zip" />
<h:inputText id="zip" size="5"
  value="#{contactController.contact.zip}">
  <f:validator validatorId="arcmind.zipCode"/>
</h:inputText>
<h:message for="zip" errorClass="errorClass" />
```

Представление страницы JSF на стороне сервера. Класс UIViewRoot.

UI Component. Объект с состоянием, методами, событиями, который содержится на сервере и отвечает за взаимодействие с пользователем. По сути, это визуальный компонент. Самое главное, что каждый UI компонент содержит метод для прорисовки самого себя — метод render. Render прорисовывает себя согласно правилам. Какие правила могут быть?
Правила задаются следующим классом — Renderer
Renderer - Отвечает за отображение компонента и преобразование ввода пользователя. То есть когда срабатывает предыдущий метод render, он обращается к Renderer и говорит: «рисуй, вот тебе мои данные, рисуй».
Validator - Проверяет правильность введенного пользователем значения. Пользователь вводит значения, они попадают на сервер и первым их хватает Validator.
Convertor - Преобразует свойства компонента в/из строки для отображения. Отвечает за преобразование поступивших данных в данные, которые понимает UI компонент. Также это правило работает и в обратную сторону.
Backing bean - Специальный JavaBean (java класс), который собирает значения из компонент, реагирует на события, взаимодействует с бизнес-логикой. Связан с каждым компонентом. Их может быть несколько, он является необязательным для UI компонента, он просто его использует.
Events and Listeners - компоненты генерируют события, слушатели реагируют на них.
Messages - сообщения, генерируемые любым объектом JSF и отображаемые пользователям . Это объект, который содержит в себе сообщение, которое будет выведено пользователю, либо не будет выведено.
Navigation - Схема навигации между страницами . Это правила навигации, которые определяют, как мы будем переходить между различными страницами. Правила задаются в виде xml документа.
JSF представляет собой набор пользовательских компонентов в виде дерева, которые формируют представление. Представление представлено объектом UIViewRoot, который связан с активным FacesContext. Состояние представления может сохраняться как на клиентской стороне (в hidden полях), так и на серверной (по умолчанию). Во время выполнения, 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 описывает с помощью 6 упорядоченных процессов, которыми можно управлять (вызывая например метод *renderResponse()* у активного *FacesContext*, либо используя свойство компонентов *immediate="true"*). Каждый раз, как пользователь отправляет запрос на сервер, один либо более процессов принимают участие в его обработке, после чего формируется и отправляется ответ.

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

Управляемые бины – классы, содержащие параметры и методы для обработки данных с компонентов. Имеют набор методов get и set для получения/установки свойств. Используются для обработки UI и валидации данных. ЖЦ 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.
Managed bean - это обычный Java бин, который зарегистрирован в JSF и управляется JSF платформой. В JSF *managed bean* используются в качестве модели для компонентов и имеют свою область жизни (*scope*), которую можно задать, как при помощи аннотации, так и в конфигурационном файле *faces-config.xml*.

Конфигурация управляемых бинов

Способ 1 — через faces-config.xml:	Способ 2 (JSF 2.0) — с помощью аннотаций:

<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>
	@ManagedBean(name="customer")
	@RequestScoped
	public class CustomerBean {
	...
	@ManagedProperty(value="#{initParam.defaultAreaCode}"
	name="areaCode")
	private String areaCode;
	...
	}

Рассмотрим варианты областей жизни, которые задаются в *managed bean* при помощи аннотаций:
@RequestScoped - используется по умолчанию. Создаётся новый инстанс *managed bean* на каждый HTTP запрос. Если, например форма будет содержать данные, которые необходимо будет отправить на сервер для обработки, то инстанс данного бина будет создаваться 2 раза: 1 - создаётся по первому запросу (*initial request*), 2 - создаётся по отправке формы (*postback*). Контекст —запрос.
@SessionScoped - инстанс создаётся один раз при обращении пользователя к приложению, и используется на протяжении жизни сессии.*Managed bean* обязательно должен быть Serializable. Контекст — сессия.
@ApplicationScoped - инстанс 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 - инстанс создаётся один раз при обращении к странице, и используется ровно столько, сколько пользователь находится на странице (включая ajax запросы). Контекст — страница.
@CustomScoped(value="#{someMap}") - инстанс создаётся и сохраняется в Map. Программист сам управляет областью жизни
@NoneScoped - инстанс создаётся, но не привязывается ни к одной области жизни. Полезно применять в *managed bean*'е, на который ссылаются другие *managed bean*'ы, имеющие область жизни
flash scope - объекты внутри этой области жизни будут доступны для последующего запроса, после чего очищаются. Другими словами объект во *flash scope* переживёт *redirect*, после чего умрёт

Конфигурация JSF-приложений. Файл faces-config.xml. Класс FacesServlet.
Конфигурирование Faces с помощью faces-config.xml
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
web.xml - стандартный конфигурационный файл, который представляет собой ядро Java web приложения
Файл faces-config.xml, содержащий объявления управляемых объектов JavaBean

```
<?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 состоит из двух частей: имени объекта — calculator —, задаваемого с помощью элемента <managed-bean-name>, и полного имени класса (элемент <managed-bean-class>). При этом класс управляемого объекта обязан содержать конструктор без параметров.
Кроме вышеперечисленных элементов существует еще один - <managed-bean-scope>, который определяет, где JSF будет искать объект. В данном случае это request. Если объект привязан к представлению (как будет показано ниже) и еще не существует на момент обращения, то JSF создаст его автоматически с помощью API универсального языка выражений EL. Все объекты, помещаемые в request, доступны только в течение обработки одного запроса. Как правило, туда имеет смысл помещать объекты, чье состояние не представляет интереса после окончательного отображения страницы в конце обработки запроса.

По умолчанию сервлет Faces будет пытаться использовать конфигурационный файл faces-config.xml, расположенный в директории WEB-INF Web-приложения. При этом можно использовать дополнительные конфигурационные файлы. Для этого предусмотрен специальный параметр в web.xml под названием javax.faces.application.CONFIG_FILES , в котором можно перечислить имена файлов, разделенные запятыми. Дополнительные файлы используются практически всегда за исключением простейших приложений.

Файл web.xml

```
<servlet>
<servlet-name>Faces Servlet</servlet-name>
<servlet-class>javax.faces.webapp.FacesServlet</servlet-class>
<load-on-startup>1</load-on-startup>
</servlet>
<servlet-mapping>
<servlet-name>Faces Servlet</servlet-name>
<url-pattern>*.jsf</url-pattern>
</servlet-mapping>
<servlet-mapping>
<servlet-name>Faces Servlet</servlet-name>
<url-pattern>/faces/*</url-pattern>
</servlet-mapping>
```

FacesServlet

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

Навигация в JSF-приложениях.

JSF включает в себя механизм навигации, аналогичный Struts. Он определяет связь между логическим признаком результата и следующим представлением. Реализуется экземплярами класса NavigationHandler.

В некоторых случаях навигация не нужна. Навигация нужна для переходов между представлениями. Но многие библиотеки компонентов JSF содержат иерархические (древовидные) представления или представления с закладками.

Навигация осуществляется с помощью правил перехода.
Добавление ссылки с домашней страницы на главную страницу
Ссылку можно добавить тремя различными способами:

С помощью commandLink и обычного правила перехода

Правило перехода, определенное в файле 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>
ИЛИ
<navigation-rule>
<from-view-id>/pages/inputname.jsp</from-view-id>
<navigation-case>
<from-outcome>sayHello</from-outcome>
<to-view-id>/pages/greeting.jsp</to-view-id>
</navigation-case>
<navigation-case>
<to-view-id>/pages/goodbye.jsp</to-view-id>
</navigation-case>
</navigation-rule>
Пример перенаправления на другую страницу:
<h:commandButton id="submit"
action="sayHello" value="Submit" />
```

Использование <h:commandLink> на домашней странице

```
<h:form>
   <h:panelGrid columns="1">
   <h:commandLink action="CALCULATOR" value="Calculator Application"/>
```

С помощью commandLink и правила перехода, использующего элемент <redirect>

Все работает, как и должно: по данной ссылке загружается главная страница Калькулятора. Однако пользователя может смутить то, что в адресной строке браузера по-прежнему фигурирует http://localhost:8080/calculator3/home.jsf. Для кого-то это может показаться нормальным, особенно для людей, часто работающих с Web-приложениями. Но это может помешать сохранить страницу Калькулятора в закладках браузера, т.к. ее подлинный адрес неизвестен.

Это можно поправить с помощью элемента <redirect> в правиле перехода, описанном в faces-config.xml

Правило перехода и элемент redirect

```
<navigation-rule>
   <navigation-case>
      <from-outcome>CALCULATOR_REDIRECT</from-outcome>
      <to-view-id>/pages/calculator.jsp</to-view-id>
      <redirect/>          <!-- LOOK HERE -->
   </navigation-case>
</navigation-rule>
```

<h:commandLink> по-прежнему использует правило перехода, используя атрибут action. Теперь при нажатии на ссылку браузер будет перенаправлен на страницу Калькулятора.

Правило перехода с использованием <redirect>

```
<h:commandLink action="CALCULATOR_REDIRECT" value="Calculator Application (redirect)"/>
```

С помощью outputLink

Таким образом, проблема решена за счет дополнительного обращения к серверу, что может занимать какое-то время в условиях медленного соединения. Однако если вы разрабатываете приложения для работы в Интранет, то это не должно стать проблемой. Этого дополнительного запроса можно избежать, если вы не против разместить прямую ссылку на требуемую страницу.

Связывание с помощью прямой ссылки (элемента <h:outputLink>)

```
<h:outputLink value="pages/calculator.jsf">
   <h:outputText value="Calculator Application (outputlink)"/>
</h:outputLink>
```

Показана прямая ссылка на следующее представление, что, как правило, считается признаком плохого стиля при использовании Model 2 и JSF. Желательно, чтобы у контроллера была возможность инициализировать классы модели для следующего представления, поэтому лучше делать не прямой вызов, а через какой-либо action-метод контроллера.

Переход на страницу результата

После выполнения арифметической операции, пользователь должен быть перенаправлен на страницу с результат.

Правило для перехода из любого представления Калькулятора на страницу результата

```
 <navigation-rule>
   <display-name>Calculator View</display-name>
   <from-view-id>/pages/calculator.jsp</from-view-id>
   <navigation-case>
      <from-outcome>results</from-outcome>
      <to-view-id>/pages/results.jsp</to-view-id>
   </navigation-case>
</navigation-rule>
```

Согласно данному правилу, если текущим представлением является calculator.jsp, а action-метод возвращает признак results, то должен быть выполнен переход к странице результатов (results.jsp). JSF автоматически преобразовывает возвращаемые значения action-методов к строковому виду и использует их для выбора следующего представления.

Пример кнопки для возвращения на домашнюю страницу

```
<div>
  <h:commandButton action="#{calculatorController.add}" value="Add" />
  <h:commandButton action="#{calculatorController.multiply}" value="Multiply" />
  <h:commandButton action="#{calculatorController.divide}" value="Divide" />
  <h:commandButton action="#{calculatorController.clear}" value="Clear" immediate="true"/>
  <h:commandButton action="HOME" value="Home" immediate="true"/>
</div>
```

Отметьте, что значением атрибута value является HOME. В листинге 47 показано правило перехода, связывающее значение HOME с домашней страницей приложения:

Правило перехода на домашнюю страницу

```
  <navigation-rule>
   <navigation-case>
      <from-outcome>HOME</from-outcome>
      <to-view-id>/home.jsp</to-view-id>
      <redirect/>
   </navigation-case>
</navigation-rule>
```

Как видите, с точки зрения правил перехода и вызова обработчиков,
элементы <h:commandButton> и <h:commandLink>работают совершенно одинаково.