**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.

Фреймворк для Web-приложений на Java, предназначенный для описания пользовательского интерфейса с помощью компонентов, находящихся на сервере

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
  + JSP (классический html + дополнительные теги)
  + Facelets (xhtml + дополнительные теги)
* Библиотека тегов
* Управляемые бины
* Доп. объекты(компоненты, конвертеры, валидаторы)
* Доп. теги
* Конфигурация – faces-config.xml
* Дискриптор развертывания – web.xml

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

**JSP** - это гораздо более старый стандарт для создания веб-страниц из шаблонов - они могут использоваться как представление в приложении JSF, но также отдельно от JSF.

**Facelets** - альтернативная технология просмотра, основанная на чистых XML-шаблонах (без скриптов), которые были представлены с версией 2 стандарта JSF. Они могут использоваться только в приложении JSF.

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.

**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 UIComponent.
* Можно создавать собственные компоненты.
* Компоненты на странице объединены в древовидную структуру — представление.
* Корневым элементов представления является экземпляр класса UIViewRoot.

Иерархия классов

-- javax.faces.component.UIComponent

---- javax.faces.component.UIComponentBase

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

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

Дополнительные теги

Предоставляют дополнительные компоненты для facelets. Примеры:

PrimeFaces, RichFaces, ICEFaces, OpenFaces, Trinidad, Tomahawk.

* F: - К базовым относятся компоненты JSF, которые не связаны с рендерингом страницы HTML или с каким-либо другим механизмом отображения информации. Среди прочего они используются для преобразования типов и проверка допустимости значений.
* H - HTML-компоненты отвечают за рендеринг разметки результирующей WEB страницы.

Для подключения – они прописываются в теге html в виде:

<html xmlns="http://www.w3.org/1999/xhtml"

xmlns:h="http://xmlns.jcp.org/jsf/html"/>

Обработка запроса

* **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
* **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.
* **Фаза валидации значений компонентов**. Вызываются валидаторы, зарегистрированные для компонентов представления. Если значение компонента не проходит валидацию, создается сообщение об ошибке и сохраняется в FacesContext.
* **Фаза обновления значений компонентов**. Если данные валидны, то значение компонента обновляется. Новое значение присваивается полю объекта компонента.
* **Фаза вызова приложения**. Управление передается слушателям событий. Формируются новые значение компонентов.
* **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.

**Конвертеры и валидаторы данных.**

Конвертеры

Когда бин привязан к UI-компоненту, существует 2 формы данных компонента

* В представлении: нечто, что может интерпретировать и вводить пользователь
* В модели: данные, хранящиеся в бине в виде значений типов данных n

Изображение выглядит как текст

Автоматически созданное описаниеКонвертеры применяются для перевода из одной формы в другую

Стандартные конвертеры JSF

* javax.faces.BigInteger
* javax.faces.Boolean
* javax.faces.Double
* javax.faces.Float

Реализация

* Автоматическая - <h:inputText value="#{user.age}"/> - age – Integer в бине
* С помощью атрибута converter: <h:inputText converter="#{javax.faces.DateTime}"/> - в теге
* С помощью вложенного тега: <h:outputText value="#{user.birthDay}">

<f:converter converterId="#{javax.faces.DateTime}"/>

</h:outputText>

Валидаторы

Когда мы уже получили данные нужного нам типа мы можем их провалидировать.

* Осуществляется перед обновлением значения компонента на уровне модели.
* Класс, осуществляющий валидацию, должен реализовывать интерфейс Validator
* Существуют стандартные валидаторы для основных типов данных.
* Можно создавать собственные валидаторы.

Изображение выглядит как текст

Автоматически созданное описаниеРеализация

* Изображение выглядит как текст

  Автоматически созданное описаниеС помощью логики на уровне управляемого бина.

**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 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.

**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 | С помощью аннотаций |
|  |  |

Изображение выглядит как текст

Автоматически созданное описаниеИзображение выглядит как текст

Автоматически созданное описание

У управляемых бинов есть контекст, который оперделяет продолжительность жизни.

* @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.

**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.

**FacesServlet** – обрабатывает запросы от браузера, формирует события и вызывает слушателей. Реализация данного фрейма позволяет нам перейти от типичного http запроса с его request – response на событие – обработчик события – FacesServlet реализует этот переход. Занимается синхронизацией состояний между представлением и DOM.

Его конфигурация задается как для обычного сервлета. Мы прописываем что все запросы будут приходить на данный сервлет.

**Навигация в JSF-приложениях.**

Навигация между страницами

* Изображение выглядит как текст

  Автоматически созданное описаниеВ файле faces-config.xml. Тут мы просто задаем navigation rule.
* Прописать в action – url нашей страницы, тогда jsf поймет что это страница при перейдет туда. <h:commandButton action = "page2" value = "Page2"/>
* Добавить в bean метод, который возвращает строку с нужным url. И поместит его в action

<h:commandButton action = "#{navigationController.moveToPage1}"

public class NavigationController implements Serializable {

public String moveToPage1() {

return "page1";

}

}

* Используя управляемый компонент, мы можем очень легко управлять навигацией

public class NavigationController implements Serializable {

@ManagedProperty(value = "#{param.pageId}")

Изображение выглядит как текст

Автоматически созданное описание private String pageId;

public String showPage() {

if(pageId == null) {

return "home";

}

if(pageId.equals("1")) {

return "page1";

}else if(pageId.equals("2")) {

return "page2";

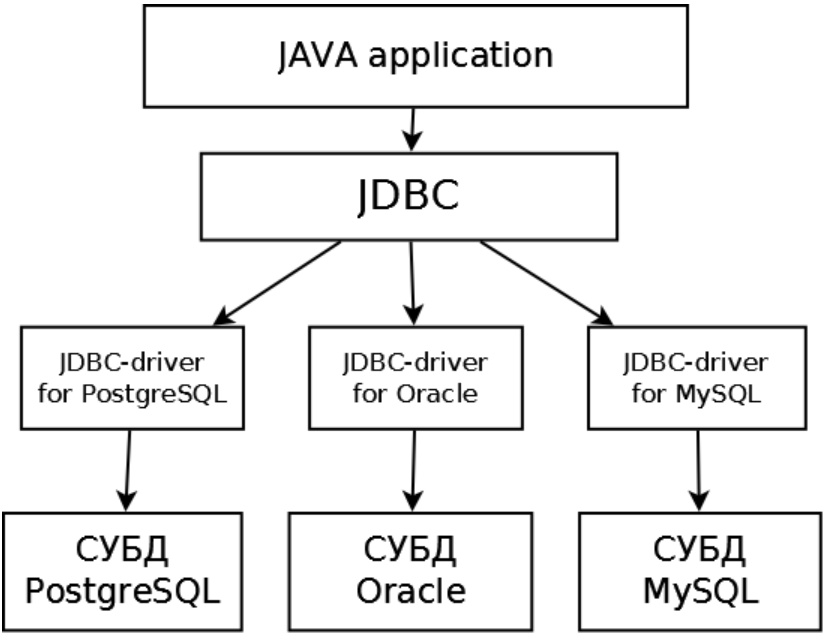
}else {

return "home";

}

}

}

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

JDBC – средство для работы с хранилищем данных. Java DataBase Connectivity – соединение с базами данных.

JDBC – набор интерфейсов и некоторых классов, которые позволяют работать с базами данных. Главный принцип архитектуры – универсальный способ общения с разными базами данных.

java.sql, javax.sql – пакеты JDBC.

Data Tier – уровень хранения данных.

Persistence Layer – некоторый уровень абстракции для работы с данными из хранилища с уровня Data Tier.

Этапы:

1) Вызов Class.forName() – загрузка класса. Например, Driver.

2) DriverManager.getConnection(url, login, password) – подключение к бд. jdbc:postgresql://localhost:5432/contactdb

Большинство классов в момент своей загрузки выполняют очень важный шаг — они РЕГИСТРИРУЮТСЯ у класса DriverManager. Первый метод как раз и позволяет классу DriverManager пройти по всему списку зарегистрированных у него драйверов и у каждого спросить — “ты умеешь работать с этим URL”. Т.е. первая часть параметра url, о которой мы говорили немного раньше, как раз и позволяет классу DriverManager выбрать драйвер для определенного типа СУБД. Первый шаг сделан — мы выбрали нужный драйвер.

И вот тут приходит очередь второго метода — именно он позволяет создать соединение — возвращает экземпляр класса, который реализует еще один важный интерфейс — java.sql.Connection.

java.sql.Connection — это реальное соединение с конкретным экземпляром СУБД определенного типа. Наше соединение готово.

JDBC - стандартный API для взаимодействия с реляционными базами данных. JDBC имеет набор классов и интерфейсов, которые могут использоваться для Java-приложения и разговаривать с базой данных.

Основные классы и интерфейсы

* DriverManager: - Это класс, использующийся для управления списком Driver (database drivers).
* Driver - Это интерфейс, использующийся для соединения коммуникации с базой данных, управления коммуникации с базой данных. Когда загружается Driver, программисту не нужно конкретно вызывать его.
* Connection - Интерфейс со всеми методами связи с базой данных. Он описывает коммуникационный контекст. Вся связь с базой данных осуществляется только через объект соединения (connection).
* Statement -Это интерфейс, включающий команду SQL отправленный в базу данных для анализа, обобщения, планирования и выполнения.
* ResultSet - представляет набор записей, извлеченных из-за выполнения запроса.

Подключение происходит так: мы скачиваем нужный нам драйвер, вызвав class.forName() мы загружаем наш драйвер и он регистрируется у DriverManager, далее с помощью DriverManager.getConnection мы создаем соединение с нашей БД.

**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.

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

**EclipseLink** 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** 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.

**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