

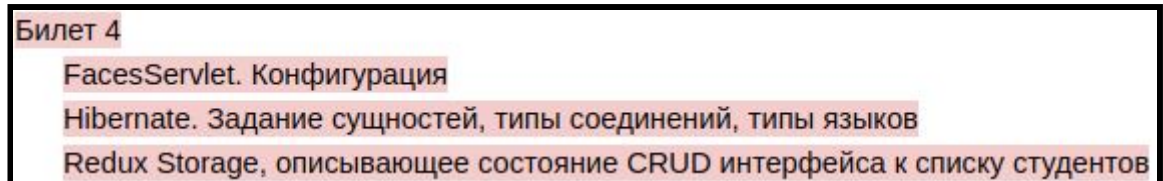
ПИП, Рубеж №2 -> Экзамен

Полезные материалы

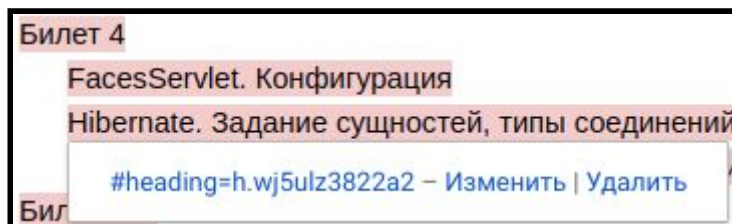
- Конспект лекций
<https://se.ifmo.ru/documents/10180/11512/Программирование+интернет-приложений/a3df4f4d-de69-490e-a64a-ec398cf4f951?version=1.14>
- Spring
<https://se.ifmo.ru/documents/10180/11512/Spring+Framework/243a4927-36a3-4659-b527-9badc67329dd?version=1.0>
- SPA, ReactJS, ES6, NPM, Babel, Webpack, Redux, Middleware
<https://se.ifmo.ru/documents/10180/11512/React+и+его+друзья/509d9908-6dee-442b-90b7-b3b31a3d3973?version=1.1>
- Angular
<https://se.ifmo.ru/documents/10180/11512/Лекция+по+Angular/6c1236d6-c1a9-4fa6-b0e6-cbebf51f583?version=1.0>
- Некоторые практические задания без гарантии достоверности
https://picloud.pw/media/resources/posts/2018/02/20/Практические_задания_ПИП.pdf
- Лекция по основам JSF от Гаврилова
<https://picloud.pw/media/resources/posts/2018/02/20/JSPJSF.pdf>

Краткое содержание

(клик на нужную главу -> клик по ссылке
например (демонстрация в скриншотах):



->клик на "Hibernate. Задание сущностей"



->появилась ссылка #heading=h.wj5ulz3822a2
клик на неё и переходишь к нужному билету)

Билеты 2017-го

Вопросы с se.ifmo.ru

Лаба №3

Лаба №4

Содержание

Билеты 2017-го	7
Билет 2	7
Структура JSF	7
Наследование и полиморфизм в ORM	7
EJB калькулятор для 4-х операций	9
Билет 3	10
Контекст управляемых бинов. Конфигурация контекста.	10
Связи между сущностями в JPA.	10
Реализовать компонент на React, строку поиска с автодополнением. Массив слов для автодополнения получать через GET запрос с использованием REST API.	10
Билет 4	10
FacesServlet. Конфигурация	10
Hibernate. Задание сущностей, типы соединений, типы языков	10
Redux Storage, описывающее состояние CRUD интерфейса к списку студентов	10
Билет 10	10
Валидаторы (все что в слайдах)	10
Запросы к бд в JPA. jpql и criteria api	10
Написать компонент ejb для списания средств со счета клиента и начисления их на счет банка за одну транзакцию	10
Билет 13	11
Фаза обновления значений компонентов, фаза вызова приложения	11
ЖЦ spring- приложения	11
Бэкенд для магазина на GWT RCP(!). Типа поля с названием, ценой, кол-вом и т.д.	11
Билет 18	11
JNDI	11
React js	11
Написать бин который выводит количество минут, прошедших со старта сервера	11
Билет 23	13
Обращение к session bean из managed и unmanaged кода	13
Структура в react. Jsx.(может как-то иначе сформулировано)	13
JSF Managed Bean, после инициализации HTTP-сессии формирующий коллекцию с содержимым таблицы Н_УЧЕБНЫЕ_ПЛАНЫ. Для доступа к БД необходимо	13

Содержание

использовать JDBC-ресурс jdbc/OrbisPool.	13
Билет 32	13
Проблемы ORM	13
Vaadin и gwt, сходства и различия	13
Написать конфиг JSF страницы которая принимает xhtml запросы и все, чей url начинается на /faces/	13
	14
Билет ?	14
JTA	14
React Router	14
Написать EJB , который " просыпается " в полночь и выводит содержимое таблицы n_люди	14
Билет ?	14
ORM	14
Don't помню(14
Servlet + html. Посчитать и вывести количество сессий в текущий момент	14
Билет ?	14
RMI. RMI в javaEE	14
инициализация Spring Beans	15
Компонент для React, реализующий интерфейс ввода данных для банковской карты: номер (16 цифр); имя (латинские символы); срок действия(mm/yy); защитный код (3 цифры)	15
Билет ?	15
JMS и реализация в Java	15
State в React redux и flux	15
JPQL запрос из таблицы, который выводит всех сотрудников старше 18 лет по полу age (Date)	15
Билет ?	15
JNDI. Способы обращения (не уверен в формулировке вопроса)	15
React.js. Ключевые особенности	15
Реализовать бин, который считает количество минут со старта приложения (или рестарта сервера); И приведён пример кода: ... <h:outputText value="#{counterBean.millisecondsAfterRestart}"/> ...	15
Билет ?	16
3 фаза жизни jsf	16
IoC DI в Spring	16
сервер JMS, который осуществляет рассылку спама всем подписчикам топика "lol"	16
Билет ?	16
1 фаза ЖЦ jsf	16
Spring, отличия от java ee	16
Сконфигурировать entity через xml	16
Билет ?	16

Содержание

Структура JSF-приложения. Компоненты JSF. Иерархия компонент	16
JPA - особенности, недостатки и преимущества, отличия от JDBC и от его использования с ORM.	16
Интерфейс на GWT, проверяющий, аутентифицирован ли пользователь. И, если нет, предлагающий ему аутентифицироваться путем ввода логина и пароля.	16
Билет ?	16
Конвертер jsf	16
Интерфейс entitymanager и его методы.	16
Виджет на gwt, для оповещений в vk, Twitter (вроде как то так звучал)	16
Билет ?	16
ajax в jsf	16
Архитектура JPA	16
Интерфейс авторизации(логин+пароль) на React. На стороне сервера - REST API	17
Билет ?	17
Управляемые бины, конфигурация и что-то еще	17
JPA	17
Написать приложение на Vaadin, которое будет автоматически заполнять приказ об отчислении.	17
Билет ?	17
JMS. Модели доставки сообщений	17
GWT. Основные преимущества и недостатки	17
Реализуйте CRUD-интерфейс к таблице Н_ВИДЫ_УЧЕБНОЙ_ДЕЯТЕЛЬНОСТИ с помощью Spring data	
Билет ?	17
Location Transparency, реализация в Java EE	17
Rest в Spring. Spring RESTful	17
Создать бин, конфигурируемый аннотациями, с именем myBean, контекст которого равен контексту другого бина - myOtherBean	18
Билет ?	18
JSF-конвертеры	18
Интерфейс EntityManager и его методы	18
Написать виджет на GWT для репостов с соцсетей (вызовы api соцсетей можно написать текстом)	18
Билет ?	18
Жизненный цикл Session beans	18
Jsx, особенности, применение в реакт	18
Написать правило навигации, для перехода с одной страницы на другую по нажатию кнопки	18
Билет ?	18
Бизнес-логика, ejb, структура и классы. Локальные и удаленные что-то.	18
SPA	18

Содержание

Что-то с БД, JPA Criteria api. Найти всех студентов со средним баллом ниже 4.0 и удалить	19
Билет 22	19
stateless stateful singleton session bean. Сходство отличия итп	19
компоненты react. Что такое итп. Умные и глупые компоненты	19
jsf поле многострочного ввода в которое можно ввести только строчные английские буквы	19
Вопросы с se.ifmo.ru	19
Лаба №3	19
Технология JavaServer Faces. Особенности, отличия от сервлетов и JSP, преимущества и недостатки. Структура JSF-приложения.	19
Использование JSP-страниц и Facelets-шаблонов в JSF-приложениях.	22
JSF-компоненты - особенности реализации, иерархия классов. Дополнительные библиотеки компонентов. Модель обработки событий в JSF-приложениях.	23
Конвертеры и валидаторы данных.	
JSF имеет встроенные конвертеры и позволяет создавать специализированные.	26
Представление страницы JSF на стороне сервера. Класс UIViewRoot.	29
Управляемые бины - назначение, способы конфигурации. Контекст управляемых бинов.	30
Конфигурация JSF-приложений. Файл faces-config.xml. Класс FacesServlet.	32
Навигация в JSF-приложениях.	33
Доступ к БД из Java-приложений. Протокол JDBC, формирование запросов, работа с драйверами СУБД.	34
Концепция ORM. Библиотеки ORM в приложениях на Java. Основные API.	
Интеграция ORM-провайдеров с драйверами JDBC.	34
Библиотеки ORM Hibernate и EclipseLink. Особенности, API, сходства и отличия.	34
Технология JPA. Особенности, API, интеграция с ORM-провайдерами.	34
Лаба №4	34
Платформа Java EE. Спецификации и их реализации.	35
Принципы IoC, CDI и Location Transparency. Компоненты и контейнеры.	36
Управление жизненным циклом компонентов. Дескрипторы развёртывания.	37
Java EE API. Виды компонентов. Профили платформы Java EE.	38
Компоненты EJB. Stateless & Stateful Session Beans. EJB Lite и EJB Full.	39
Работа с электронной почтой в Java EE. JavaMail API.	39
JMS. Реализация очередей сообщений. Способы доставки сообщений до клиента. Message-Driven Beans.	40
Понятие транзакции. Управление транзакциями в Java EE. JTA.	40
Веб-сервисы. Технологии JAX-RS и JAX-WS.	41
Платформа Spring. Сходства и отличия с Java EE.	41
Модули Spring. Архитектура Spring Runtime. Spring Security и Spring Data.	41
Реализация IoC и CDI в Spring. Сходства и отличия с Java EE.	42
Реализация REST API в Java EE и Spring.	42

React JS. Архитектура и основные принципы разработки приложений.	42
Компоненты React. State & props. "Умные" и "глупые" компоненты.	42
Разметка страниц в React-приложениях. JSX.	42
Навигация в React-приложениях. ReactRouter.	42
Управление состоянием интерфейса. Redux.	42
Angular: архитектура и основные принципы разработки приложений.	42
Angular: модули, компоненты, сервисы и DI.	43
Angular: шаблоны страниц, жизненный цикл компонентов, подключение CSS.	43
Angular: клиент-серверное взаимодействие, создание, отправка и валидация данных форм.	43

Билеты 2017-го

- **Билет 2**

1. Структура JSF

[Технология JavaServer Faces. Особенности, отличия от сервлетов и JSP, преимущества и недостатки. Структура JSF-приложения.](#)

[JSF-компоненты - особенности реализации, иерархия классов. Дополнительные библиотеки компонентов. Модель обработки событий в JSF-приложениях.](#)

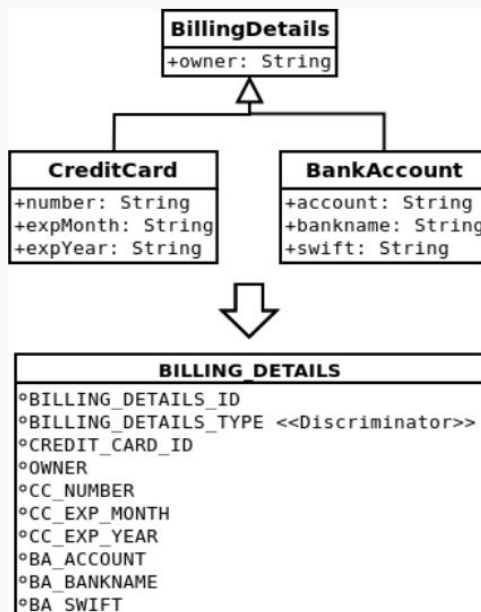
[Представление страницы JSF на стороне сервера. Класс UIViewRoot.](#)

2. Наследование и полиморфизм в ORM

Существует три способа реализации наследования в ORM



Одна таблица для иерархии классов (Single Table Inheritance Pattern)



Все классы иерархии отображаются на одну таблицу базы данных.

Достоинства:

- 1) Наиболее простое решение.
- 2) Наиболее производительное решение.

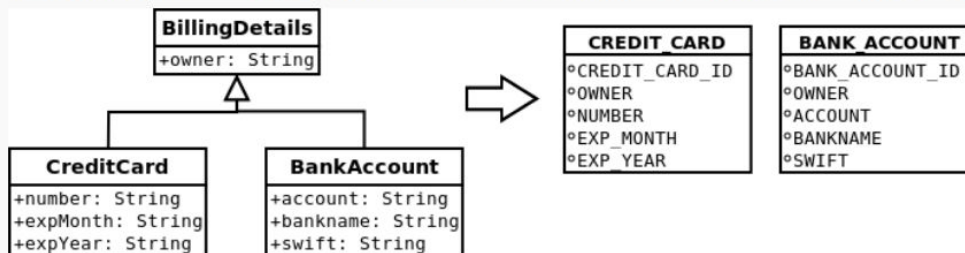
Недостатки:

- 3) На поля подклассов нельзя накладывать null-ограничения.
- 4) Полученная таблица не нормализована.



Своя таблица для конкретного класса (Concrete Table Inheritance Pattern)

Каждый класс в иерархии отображается на отдельную таблицу БД.



Достоинства:

Можно накладывать ограничения (not null) на поля подклассов.

Недостатки:

- 1) Полученные таблицы не нормализованы.
- 2) Плохая поддержка полиморфных запросов.
- 3) Низкая производительность.



3. EJB калькулятор для 4-х операций

```
@Stateless(name = "CalculatorEJB")
public class CalculatorEJB{
    public float add(float a, float b){
        return a + b;
    }
    public float sub(float a, float b){
        return a - b;
    }
    public float mul(float a, float b){
        return a * b;
    }
    public float div(float a, float b){
        return a / b;
    }
}
```

вар2(не нужно передавать аргументы в сигнатуру, а можно привязать инпуты к проптерям и запускать action)

```
@Stateless(name = "CalculatorEJB")
public class CalculatorEJB{
```



```
private double a; private double b; private double ans;
public getA(){return a;} public getB(){return b;}
public setA(double a){this.a=a;} public setB(double b){this.b=b;}
public getAns(){return ans;};
    public float add(){
        ans= a + b;
    }
    public float sub(){
        ans= a - b;
    }
    public float mul(){
        ans= a * b;
    }
    public float div(){
        ans= a / b;
    }
}
```

● **Билет 3**

1. Контекст управляемых бинов. Конфигурация контекста.

6 видов контекста:

application scope - жц управляет приложение

view scope - жц - страница

session scope - жц управляет сессия

request scope - бин живет только во время запроса

nonescaped - жц управляют другие бины

custom scope - кладем бин в мапу и управляет его жц сами

конфигурация с помощью аннотаций или в xml файле

Способ 1 — через 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>
```

Способ 2 (JSF 2.0) — с помощью аннотаций:

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

2. Связи между сущностями в JPA.

OneToMany
ManyToOne
ManyToMany
OneToOne

двунаправленные - классы хранят ссылки друг на друга

однаправленные - ссылка хранится только в одном из классов

3. Реализовать компонент на React, строку поиска с автодополнением. Массив слов для автодополнения получать через GET запрос с использованием REST bAPI.

```
import axios from 'axios'
import Autocomplete from 'react-autocomplete'

class seacrh extends React.Component{
  constructor(props) {
    super(props)
    this.state = {
      value: ''
      hints: [] //{label: 'qwe'}, {label: 'asd'} и тд
    }
    this.getItems();
  }

  getItems() {
    axios.get('wordsURL').then(
      res => {
        this.setState({hints: resdata})
      }
    )
  }

  render() {
    return(
      <textarea value={this.state.value} onChange={this.onChange.bind(this)} />
      <Autocomplete
        getItemValue={(item) => item.label}
        items={this.state.hints}
        renderItem={(item, isHighlighted) =>
          <div style={{ background: isHighlighted ? 'lightgray' : 'white' }}>
            {item.label}
          </div>
        >
        value={this.state.value}
        onChange={(e) => value = e.target.value}
        onSelect={(val) => value = val}
      />
    )
  }
}
```

- **Билет 4**

1. FacesServlet. Конфигурация

Сервлет, который будет делегировать выполнение запросов на указанных урлах jsf фреймвоку

FacesServlet создаёт объект FacesContext, который хранит информацию, необходимую для обработки запроса. FacesContext содержит всю информацию о состоянии запроса во время процесса обработки одного JSF-запроса, а также формирует ответ на соответствующий запрос.

```
<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>/faces/*</url-pattern>
</servlet-mapping>
```

2. Hibernate. Задание сущностей, типы соединений,

ТИПЫ ЯЗЫКОВ

[Hibernate](#) - [ORM](#) framework реализация [JPA](#) api, один из основных. Сущности задавать через аннотации или через persistence.xml, где прописывается различное описание сущности, например, к какой таблице относится и т.д.

Соединения сущностей бывает OneToMany, ManyToOne, OneToOne, ManyToMany

Языки:

HQL - hibernate query language

3. Redux Storage, описывающее состояние CRUD интерфейса к списку студентов

● Билет 10

1. Валидаторы (все что в слайдах

[Валидаторы](#) это элементы jsf, которые позволяют осуществлять проверки вводимых значений в компоненты jsf. Валидаторы начинают работать после фазы применения значений запроса(после того, как отработали конверторы) или если стоит `immediate="true"`. Есть готовые валидаторы длины, регулярок и т.д. можно писать свои валидаторы, которые должны реализовать интерфейс валидатор

2. Запросы к бд в JPA. jpql и criteria api

[Для доступа к БД используется класс EntityManager.](#)

```
▶ <T> T find(Class<T> entityClass, Object primaryKey)
▶ <T> T getReference(Class<T> entityClass, Object primaryKey)
▶ void persist(Object entity)
▶ <T> T merge(T entity)
▶ refresh(Object entity)
▶ remove(Object entity)
▶ void flush()
▶ void close();
```

Ещё там есть `createQuery`, `createNativeQuery` и тому подобное.

Для выполнения запросов используются SQL, JPQL (Java Persistence Query Language - SQL-подобный язык для работы с объектами вместо сущностей) и Criteria API (не SQL-подобные текстовые запросы, а методы).

Пример:

```
EntityManagerFactory emfactory = Persistence.
    createEntityManagerFactory("Eclipselink_JPA");

EntityManager entitymanager = emfactory.
    createEntityManager();

//Scalar function
Query query = entitymanager.
```

```
createQuery("Select UPPER(e.ename) from Employee e");  
List<String> list=query.getResultList();  
Query query = entityManager.  
createQuery("Select e.ename from Employee e");  
List<Employee> eList=query.getResultList();
```

3. Написать компонент ejb для списания средств со счета клиента и начисления их на счет банка за одну транзакцию

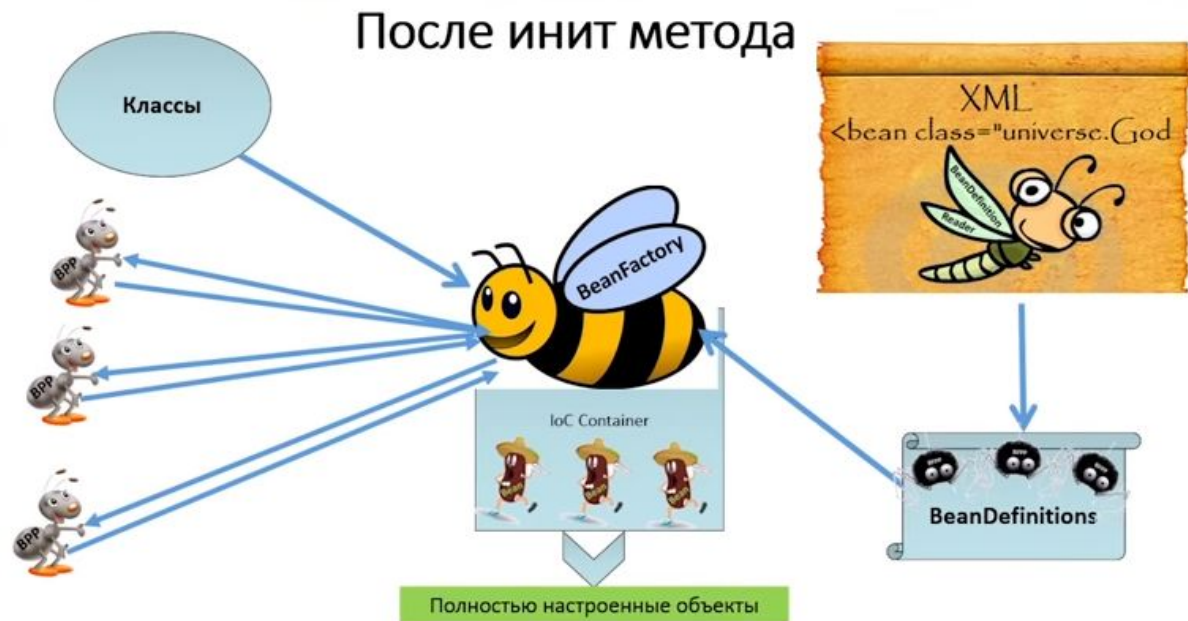
```
@Stateless  
@TransactionManagement(BEAN)  
public class MyBean {  
    @Resource  
    UserTransaction ut;  
    public void pay(Bank bank, Client client, double sum) {  
        try {  
            ut.begin();  
            client.setWalletValue(client.getWalletValue - sum);  
            bank.setWV(bank.getWV+sum);  
            ut.commit();  
        } catch (Exception e) {  
            ut.rollback();  
        }  
    }  
}
```

● **Билет 13**

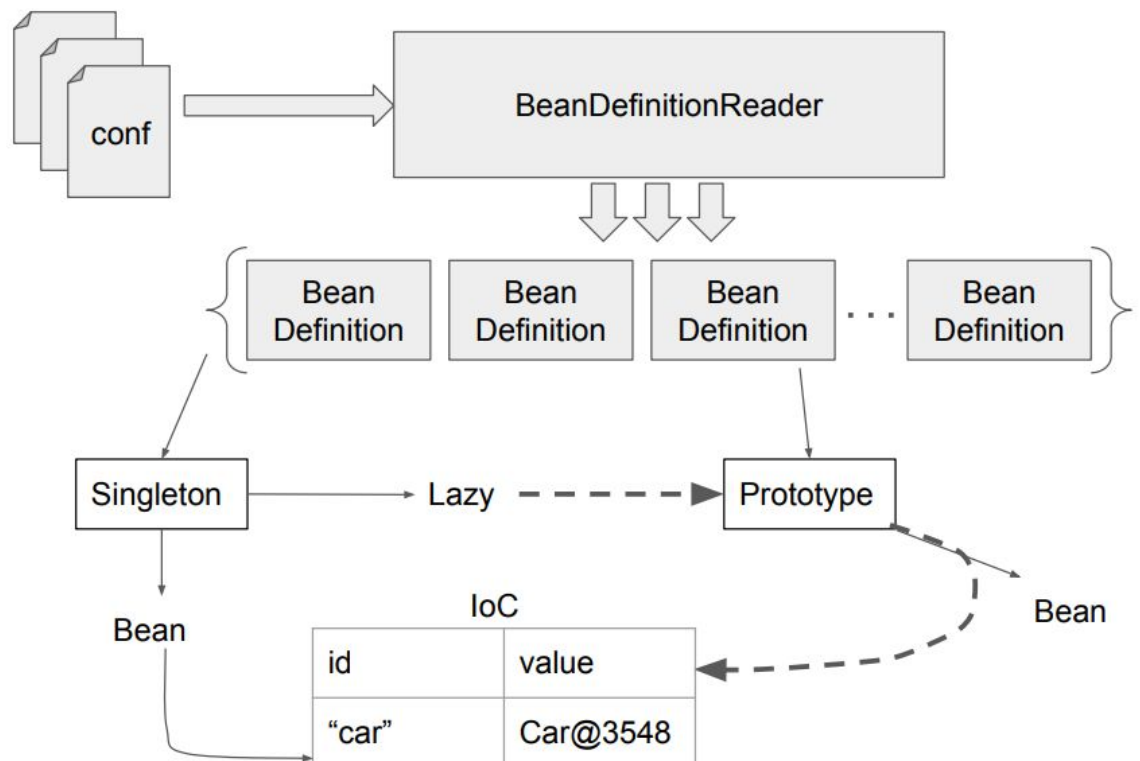
1. Фаза обновления значений компонентов, фаза вызова приложения

На фазе обновления значений компонентов уже отработали валидаторы, значит данные условно валидны и мы передаем эти данные в проперти бина. В процессе обновления мы можем исполнить евенты, на которые кто то подписан. если event listener запустил responseComplete(), то обработка запроса прекращается, если запустили renderResponse() обработка уходит на стадию формирования ответа. На фазе вызова приложения оставшиеся события передаются для обработки приложению

2. Жизненный цикл spring- приложения



Жизненный цикл Spring-приложения (упрощенный)



beandefinition reader парсит конфиг, (разные типы ридеров парсят разные конфиги) и создает beandefinition это мета информация о бинах, их скоуп, название и т. д. потом создаются beanfactorypostprocessor которые имеют доступ к beandefinition, которые могут изменить

дефинишны. После этого создаются beanpostprocessor, которые могут получить доступ к классам. beanfactory вызывает конструкторы бинов, скормливает бины постпроцессорам, которые могут например распарсить аннотации и по ним создать динамические прокси от бинов и добавить им функционал, после этого фактори складывает бины в ИОС контейнер.

3. Бэкэнд для магазина на GWT RCP(!). Типа поля с названием, ценой, кол-вом и т.д.

● Билет 18

1. JNDI

Есть такая штука - JNDI - набор API, позволяющий доставать объекты из контейнера по их именам. По сути, это замена обращению по ссылкам, и благодаря этому легко реализуется Dependency Lookup и [Dependency Injection](#). Хранение объектов в таком случае можно представить как Map, где ключи - это имена, а значения - нужные объекты (например, ссылки на них).

2. React js

Библиотека для построения ui. Использует компонентный подход для создания ui. Использует jsx для построения дом модели. Использует виртуальный дом для оптимизации рендера компонентов. (не обновляет дом целиком, а обновляет виртуальный дом и сравнивает, что нужно обновить в текущем состоянии дома). есть умные и глупые компоненты. у умных компонентов можно управлять состоянием и методами жизненного цикла компонента

3. Написать бин который выводит количество минут, прошедших со старта сервера

```
@ManagedBean
@ApplicationScoped
public class CounterBean implements Serializable{
    long startTime;
    public CounterBean() {
        startTime = System.currentTimeMillis();
    }
    public long getMinutesAfterRestart() {
        return (System.currentTimeMillis() - startTime) / 1000 / 60;
    }
    public void setMinutesAfterRestart() {}
}
```

- **Билет 23**

1. Обращение к session bean из managed и unmanaged кода

Managed-код находится внутри managed компонентов, то есть компонентов, которыми управляет / о которых знает контейнер, то есть над которыми применимы IoC и CDI. Соответственно, обращение к SessionBean из managed-кода осуществляется с помощью CDI (аннотация @EJB над полем, в котором должен лежать этот бин), а из unmanaged - с помощью JNDI. Примеры:

```
@EJB(name="beanName", beanInterface = Bean.class)
```

```
Bean beanInstance = (Bean) new  
InitialContext().lookup("java:comp/env/beanName");
```

2. Структура в react. Jsx.(может как-то иначе сформулировано)

Структура реакт приложений - древовидная, основанная на прямом потоке данных, где родители могут что то передать детям. по этой структуре строится виртуальный дом, а затем реальный. Jsx - язык, который выглядит, как смесь js и html, мы можем помечать код как js вставку с помощью {}. Все потом парсится в js с помощью бабеля

3. JSF Managed Bean, после инициализации HTTP-сессии формирующий коллекцию с содержимым таблицы Н_УЧЕБНЫЕ_ПЛАНЫ. Для доступа к БД необходимо использовать JDBC-ресурс jdbc/OrbisPool.

```
@ManagedBean
@SessionScoped
class MyBean implements Serializable {

    @Resource(name="jdbc/OrbisPool")
    private DataSource dataSource;

    HashMap results;

    @PostConstruct
    public void init() throws SQLException {
        Connection connection = dataSource.getConnection();
        PreparedStatement ps = connection.prepareStatement("select * from Н_УЧЕБНЫЕ_ПЛАНЫ");
        ResultSet resultSet = ps.executeQuery();
        results = resultSetToHashMap(resultSet);
    }
}
```

● Билет 32

1. Проблемы ORM

[Есть три важные проблемы:](#)

- Проблема идентичности.
- Представление наследования и полиморфизма.
- Проблема навигации между данными.

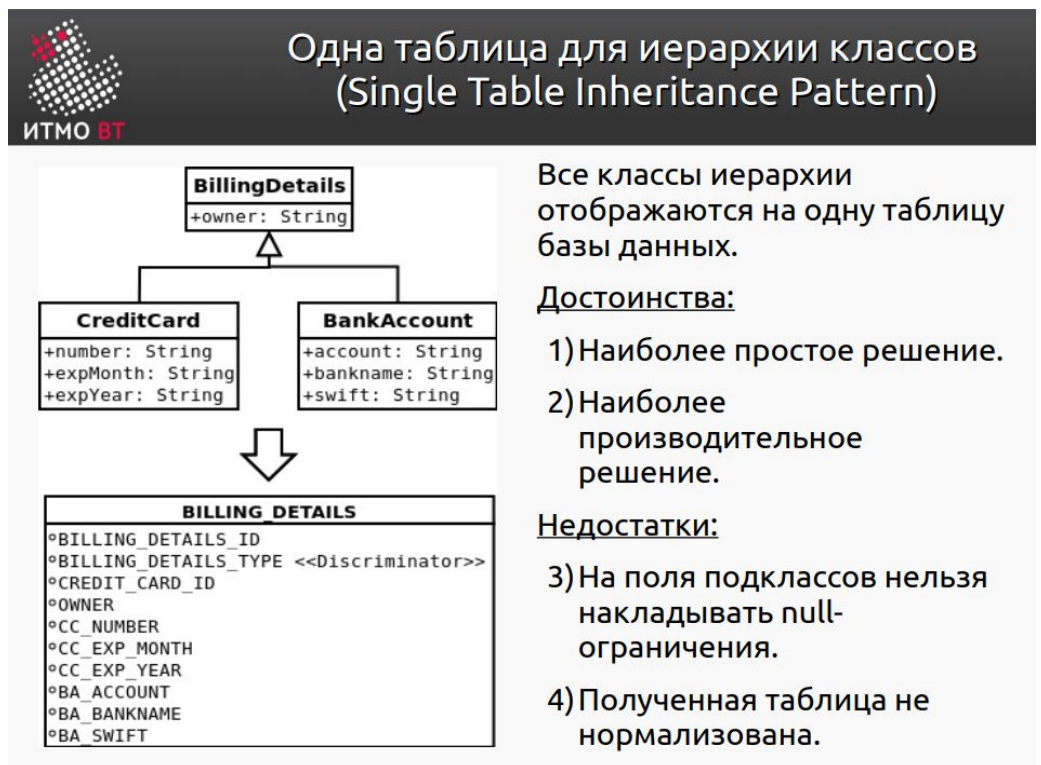
Решения первой проблемы:

Возможны три варианта:

- 1) Обычный уровень хранения без управления идентичностью (no identity scope).
- 2) Уровень хранения с контекстно-управляемой (context-scoped) идентичностью .
- 3) Уровень хранения с жестким управлением идентичностью (process-scoped identity).

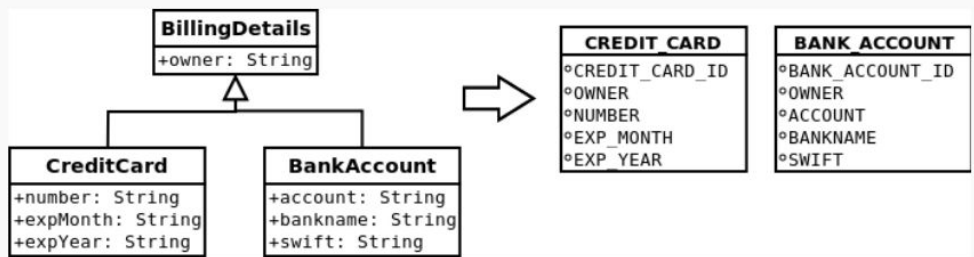
Решения проблемы наследования и полиморфизма (3 штуки):

- Одна общая таблица



- По таблице на каждого наследника

Каждый класс в иерархии отображается на отдельную таблицу БД.



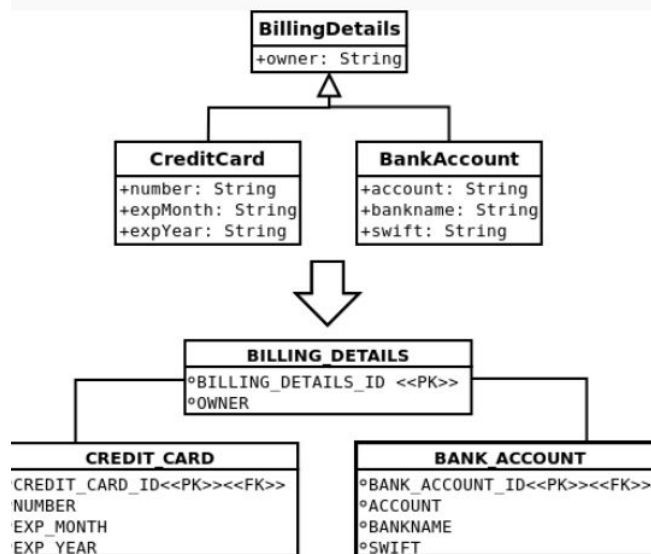
Достоинства:

Можно накладывать ограничения (not null) на поля подклассов.

Недостатки:

- 1) Полученные таблицы не нормализованы.
- 2) Плохая поддержка полиморфных запросов.
- 3) Низкая производительность.

- Таблицы и для предков, и для наследников



Каждый подкласс отображается на отдельную таблицу, которая содержит колонки, соответствующие только полям этого подкласса.

Достоинства:

- 1) Таблицы нормализованы.
- 2) Лучший вариант для полиморфизма.
- 3) Существует возможность задавать ограничения на поля подклассов.

Недостатки:

Запросы выполняются медленнее, чем при использовании одной таблицы.

2. Vaadin и gwt, сходства и различия

3. Написать конфиг JSF страницы которая принимает xhtml запросы и все, чей url начинается на /faces/

```
<web-app>
  <servlet>
    <servlet-name>Faces Servlet</servlet-name>
    <servlet-class>javax.faces.webapp.FacesServlet</servlet-class>
  </servlet>
  <servlet-mapping>
    <servlet-name>Faces Servlet</servlet-name>
    <url-pattern>*.xhtml</url-pattern>
  </servlet-mapping>
  <servlet-mapping>
    <servlet-name>Faces Servlet</servlet-name>
    <url-pattern>/faces/*</url-pattern>
  </servlet-mapping>
</web-app>
```

● Билет ?

1. JTA

[java transaction api](#) - дает возможность управления транзакцией не на уровне орм, а на уровне жава кода. инжектится юзертранзакцион от него можно делать бегин, коммит, ролбэк

Нужно для того, чтобы, когда наше приложение работает сразу с нескольких JVM (и юзуются RemoteBean'ы), транзакции распространялись на все JVM, то есть сохраняли изолированность.

2. React Router

Все приложение надо обернуть в <BrowserRouter> - если приложение нестатическое (обрабатываются динамические запросы)
Каждый Router создает объект history который хранит путь к текущему location[1] и перерисовывает интерфейс сайта когда происходят какие то изменения пути. Остальные функции предоставляемые в React Router полагаются на доступность объекта history через context, поэтому они должны рендериться внутри компонента Router.

<Route/> компонент это главный строительный блок React Router'a.
<Route /> принимает path в виде проп который описывает определенный путь и сопоставляется с location.pathname. Компонент Route может быть в любом

месте в роутере, но иногда нужно определять, что рендерить в одно и то же место. В таком случае следует использовать компонент группирования Route'ов — `<Switch/>`. `<Switch/>` итеративно проходит по дочерним компонентам и рендерит только первый который подходит под `location.pathname`. Оборачиваем руты свитчем. Также в Route надо указать какой компонент рендерить - атрибут `component`. Рендериться будут все вариации url'ов, содержащие `path`, если не указать `exact`. Тогда будет рендериться строго переданный url.

3. Написать EJB , который " просыпается " в полночь и выводит содержимое таблицы н_люди

из доков по аннотации `@Schedule`: All elements of this annotation are optional. If none are specified a persistent timer will be created with callbacks occurring every day at midnight in the default time zone associated with the container in which the application is executing.

То есть достаточно просто поставить эту анноташку над нужным методом и магия свершится

`@ApplicationScoped`

`@Stateful` // используется инъекция, все дела

`@LocalBean` // а почему нет?

```
public class Bessonnitsa {
    @PersistenceContext(unitName = "movie-unit",
        type = PersistenceContextType.EXTENDED)
    private EntityManager em;

    @Schedule
    public void doParty() {
        for (NLudiEntity human :
            em.createQuery("SELECT h from н_люди h")
                .getResultList())
            System.out.println(human);
    }
}
```

● Билет ?

1. ORM

Технология, которая позволяет связывать объектное и реляционное отображения. Существуют несколько орм-провайдеров. Самые известные - гиббернейт и эклипслинк. Использует jdbc и является высшим уровнем абстракции над jdbc.

2. Don't помню

3. Servlet + html. Посчитать и вывести количество сессий в текущий момент

Вроде как сервлетов быть не должно, но на всякий случай вот:

```
public class SessionCounter extends HttpServlet {
    private ArrayList<HttpSession> sessions = new ArrayList<>();

    protected void doPost(HttpServletRequest request,
        HttpServletResponse response)
        throws ServletException, IOException {
        boolean gonnaAdd = true;
        for (int i = 0; i < sessions.size(); i++) {
            HttpSession temp = sessions.get(i);

            if (temp.getId() != request.getSession().getId()) {
                if (System.currentTimeMillis() -
                    temp.getLastAccessedTime() >=
                    temp.getMaxInactiveInterval() * 1000L)
                    sessions.remove(i--);
            }

            gonnaAdd = false;
            sessions.replace(i, temp = request.getSession());
        }
        if (gonnaAdd)
            sessions.add(request.getSession());

        try (PrintWriter out = response.getWriter()) {
            out.print("<!DOCTYPE HTML>");
            out.print("<html><head></head><body>");
            out.print(sessions.size() + " sessions alive");
            out.print("</body></html>");
        }
    }
}
```

- **Билет ?**

1. RMI. RMI в javaEE

remote method invocation - технология, которая позволяет вызывать методы удаленных объектов. Объекты передаются по значению, а не по ссылке. Если объект экспортирован передается заглушка(stub).
Передаваемые объекты должны быть сериализуемыми

RMI тесно связан с принципом [Location Transparency](#). Пример использования RMI (и, соответственно, реализации Location Transparency): Remote EJB и Local EJB. Для клиента создаётся видимость целостности приложения, как будто оно не распределено по разным серверам с разными JVM, то есть для использования Remote EJB ему не надо делать дополнительных движений.

2. инициализация Spring Beans

Итак, поехали

У Spring есть четкий порядок инициализации объектов:

Формируется Configuration Metadata, она может быть создана из XML-контекста, из конфигурации с помощью Annotations либо Java Configuration.

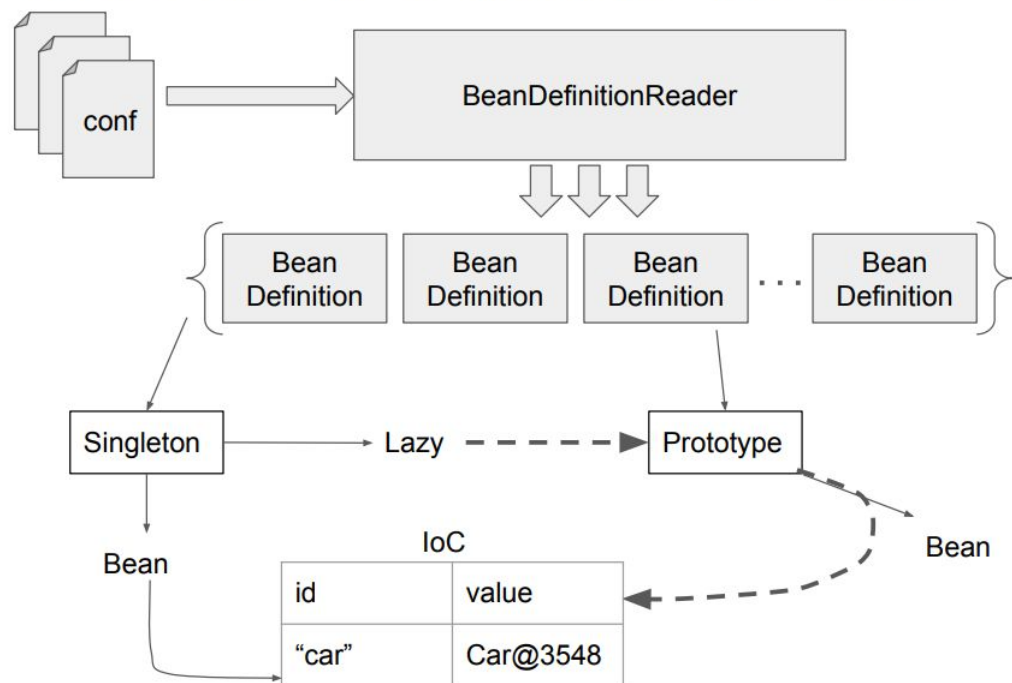
Все объекты, которые имплементируют интерфейс BeanFactoryPostProcessor, читают Metadata и изменяют ее в соответствии со своим предназначением.

Вся Metadata, которую модифицировали и нет, передается в BeanFactory, которая непосредственно и создает spring beans.

Все объекты, которые имплементируют интерфейс BeanPostProcessor, производят pre initializing- и post initialization-действия.

Все бины, которые уже были инициализированы, отдаются в IoC Container.

Жизненный цикл Spring-приложения (упрощенный)



3. Компонент для React, реализующий интерфейс ввода данных для банковской карты: номер (16 цифр); имя (латинские символы); срок действия(mm/yy); защитный код (3 цифры)

```
import React, { Component } from 'react';
import 'primereact/resources/themes/nova-light/theme.css';
import 'primereact/resources/primereact.min.css';
import 'primeicons/primeicons.css';
import { Button } from 'primereact/button';
import { InputText } from 'primereact/inputtext';
import { Calendar } from 'primereact/components/calendar/Calendar';

class Bank extends React.Component {
  constructor(props) {
    super(props);
    this.state = {
      number: '',
      name: '',
      date: '',
      code: ''
    };
  }

  handleChange = name => event => {
    this.setState({
      [name]: event.target.value,
    });
  };
}
```

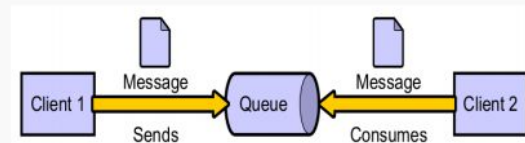
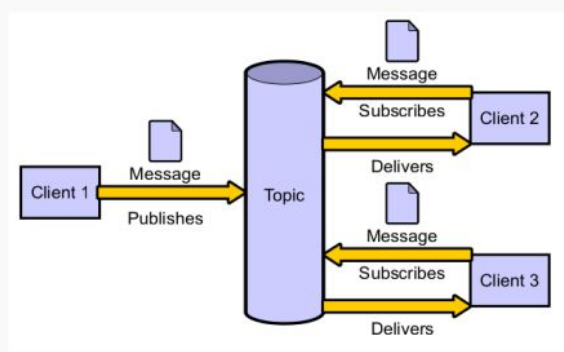
Содержание

```
};
clickButton = () => {
  //...обработка нажатия кнопки
};
render() {
  return (
    <div>
      <form id="formLogIn" >
        <h1>Введите данные карты:</h1>
        <h3>Номер:</h3>
        <InputText maxLength="16" keyfilter={/[0-9^\s]/}
value={this.props.number} onChange={this.handleChange('number')}/>
        <h3>Имя:</h3>
        <InputText keyfilter={/[A-z^\s]/} value={this.props.name}
onChange={this.handleChange('name')}/>
        <h3>Срок действия:</h3>
        <Calendar value={this.state.date}
onChange={this.handleChange('date')} view="month" dateFormat="mm/yy"
yearNavigator={true} yearRange="2010:2030"/>
        <h3>Защитный код:</h3>
        <InputText maxLength="3" keyfilter={/[0-9^\s]/}
value={this.props.code} onChange={this.handleChange('code')}/>
        <Button label="Send" onClick={this.clickButton}/>
      </form>
    </div>
  );
}
}
export default Bank;
```

- **Билет ?**

1. JMS и реализация в Java

- Позволяет организовать асинхронный обмен сообщениями между компонентами.
- Две модели доставки сообщений — «подписка» (topic) и «очередь» (queue):



На самом деле можно и синхронно (то есть получатель вызывает метод receive, и ему либо даётся Message, либо null).

Есть три важные штуки: Session, MessageProducer и MessageConsumer. Вся схема с доставкой / получением сообщений будет работать только пока сессия запущена. Она может быть приостановлена методом stop и возобновлена start. Методами сессии создаются поставщики и потребители. Для каждого из них можно передать в конструктор куда они будут обращаться (topic или queue), но если этого не сделать, то при отправке / получении сообщения можно будет указать куда / откуда это делать. С поставщиком всё просто: формирует сообщение (разные для разных типов контента) и вызывает producer.send(msg). С потребителем интереснее: он может сделать receive (синхронный способ, может выдавать null), а можно навесить на этого потребителя MessageListener, который будет вызываться как только сообщение будет доставлено, а не когда потребитель этого захочет, то есть асинхронно. Теперь про разницу между Topic и Queue. В случае топика поставщик - это что-то типа блоггера, он пишет сообщение, а подписота читает. Однако у сообщений есть тайм-аут, после которого их уже нельзя будет получить. В случае Queue сообщение исчезнет как только какой-либо потребитель его получит, то есть гарантируется, что только один юзер получит его. Таймаута на сообщении нет. Юзер может отказываться от сообщения, чтобы следующий в очереди мог посмотреть, не ему ли оно

2. State в React redux и flux

Общее состояние для всего приложения. Хранится в Redux store, не должно меняться никак кроме использования метода dispatch, который принимает reducer. Reducer - функция, которая принимает старое состояние и действие(объект с данными, которые меняем - опционально и типом действия - type - обязательно) и возвращает новое состояние в зависимости от типа действия. Умные компоненты могут получать доступ к значениям стейтам и редюсерам с помощью метода connect, который используя прописанные в компоненте функции mapStateToProps, mapDispatchToProps. Редюсеры также позволяют работать не со всем стейтом, а с его частями, объединяя их в одно с помощью метода combineReducers. Просто получить значение стейта - getState(). Узнать об изменениях в стейте - subscribe(listener) - вызывается всякий раз, когда был вызван dispatch, принимает функцию, которая будет вызвана, возвращает функцию, которая отпишет слушателя.

3. JPQL запрос из таблицы, который выводит всех сотрудников старше 18 лет по полу age (Date)

```
@Resource EntityManager em  
List<Employee> list = em.createQuery("Select e from Employee e where  
e.age > 18").getResultList();
```

● Билет ?

1. JNDI. Способы обращения (не уверен в формулировке вопроса)

JNDI - набор API, позволяющий доставать объекты из контейнера по их именам. По сути, это замена обращению по ссылкам, и благодаря этому легко реализуется Dependency Lookup и [Dependency Injection](#). Хранение объектов в таком случае можно представить как Map, где ключи - это имена, а значения - нужные объекты (например, ссылки на них).

Пример: `WtfBean example = new InitialContext().lookup("wtf_bean's name");`
Контекст - это, как ни странно, контекст приложения - необходимая часть контейнера, в котором используются IoC и CDI.

2. React.js. Ключевые особенности

Использование виртуального дома(оптимизация, не нужно каждый раз перерендеривать дом), реактивное связывание компонентов. Все построено на компонентах(умных - классы, можно переопределять методы, управлять жц, есть состояние и глупых - функциональные,

только рендер, нет состояния) и их иерархии. Однонаправленный поток данных - передача данных от родителей к потомкам через props

Умный: class comp extends React{

constructor(props){// конструктор, можно задавать изначальное состояние, обязательно вызывать супер

super(props)

this.state = {

id: 0,

name: PipSucks

}

}

//переопределение методов реакта типа

componentDidMount, componentShouldUpdate или свои методы для обработки событий (еще примеров есть в заданиях)

render(){

return(...//Элемент, хтмл код со всякими вставками if

и тд

<CustomElem

parentName={this.state.name}>Hello</CustomElem >

//передача в пропсы потомка чего-нибудь из состояния компонента

)

}

}

Глупый:

function Hello(props){

return <h2>Hello, {props.name}</h2>

}

const element = <Hello name="qwerty"/>

3. Реализовать бин, который считает количество минут со старта приложения (или рестарта сервера); И приведён пример кода: ... <h:outputText value="#{counterBean.millisecondsAfterRestart}"/> ...

@ManagedBean

@ApplicationScoped

public class CounterBean implements Serializable{


```
long startTime;  
public CounterBean() {  
    startTime = System.currentTimeMillis();  
}  
public long getMillisecondsAfterRestart() {  
    return (System.currentTimeMillis() - startTime) / 1000 / 60;  
}  
public void setMillisecondsAfterRestart() {}  
}
```

● Билет ?

1. 3 фаза жизни jsf

[processvalidations](#). запускаются валидаторы для компонентов, которые проверяют совпадают ли данные с заранее заданными условиями. (мы не проверяем совпадают ли данные по типу это на 2 фазе, когда работают конвертеры) Если на этой фазе ловится ошибка, то остальные пропускаются и мы рендерим ответ с этой ошибкой

2. IoC DI в Spring

Бины имеют свои скопы(прототип, синглтон (+ сессия и запрос, если юзается в веб-приложении)) можно реализовать свои области видимости, бины помечаются аннотациями(@service, @component...) организация di идет с помощью @Autowired, @Inject, @Resource. можно инжектировать с помощью конструктора, можно инжектировать только какое то поле

[IoC](#) - это когда контейнер занимается порождением и управлением компонентов, и нам не приходится так часто видеть new в коде. Об управлении жизненным циклом компонентов в спринге [написано тут](#).

3. сервер JMS, который осуществляет рассылку спама всем подписчикам топика "lol"

```
// Jms не будет скорее всего  
@Resource(mappedName="jms/ConnectionFactory")  
private static ConnectionFactory connectionFactory;  
@Resource(mappedName="lol")private static Topic topic;  
public void sendSpam() {  
    Connection connection = connectionFactory.createConnection();  
    try {
```

```
Session session = connection.createSession(false,
        Session.AUTO_ACKNOWLEDGE);
MessageProducer producer = session.createProducer(topic);
connection.start();

TextMessage message = session.createTextMessage();
message.setText("даже школьник смог написать сайт
просто прочитав этот ...");

while (true) {
    producer.send(message);
    Thread.sleep(1000 * 60);
}

catch (InterruptedException e) {
    e.printStackTrace();
} finally {
    if (connection != null) {
        try { connection.close(); }
        catch (JMSException e) {e.printStackTrace();}
    }
}

}
```

● Билет ?

1. 1 фаза ЖЦ jsf

- 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.

2. Spring, отличия от java ee

Во первых, приложение Spring это самостоятельное приложение, которому для работы ничего не нужно, так как оно не связано ни с каким контейнером, а сервер, если он есть, то есть приложение серверное, в него уже встроен. Так же вполне себе может функционировать и без сервера и вообще не принадлежать к миру WEB. В отличие от Java EE, она и предназначена для → серверных приложений ← , она предоставляет целый контейнер, кластер java ee, который дает много возможностей из коробки, даже слишком много. Из плюсов, много возможностей. Из минусов, прозрачность процессов идет к черту.

java ee базируется на трехуровневой архитектуре, есть бизнес тир, есть клиент тир, есть вью тир. Спринг базируется на куче независимых модулей. Java ee based on high-level language. Spring не зависит от языка.

3. Сконфигурировать entity через xml

```
<entity class="User" name="User">
  <table name="User"/>
  <attributes>
    <id name="Username"/>
    <basic name="dolzhnost"/>
    <basic name="password"><column="pswd"
length=50/><basic>
  </attributes>
</entity>
```

● Билет ?

1. Структура JSF-приложения. Компоненты JSF.

Иерархия компонент

древовидная структура jsf приложения. [Компоненты jsf](#) связаны с их рендерером, связаны с компонентом message, который может отображать сообщения об успехе/провале. компонент может генерировать события, на которые подписываются слушатели. особый тип события - action может генерироваться элементами типа button. он возвращает строку, которая потом может служить ключом для правил навигации. с компонентом связаны валидатор и конвертер, которые служат для проверки и

трансляции данных из строки в нужный формат. Корнем иерархии компонентов служит `uiViewwroot`.

2. JPA - особенности, недостатки и преимущества, отличия от JDBC и от его использования с ORM.

[jpa это спецификация для работы с базами данных](#), а jdbc это стандарт. они нужны для работы с базами данных. jpa использует орм провайдер, который использует ждбс, который [использует ждбс драйвер](#) для доступа в базу данных. jpa позволяет абстрагироваться от особенностей орм провайдера и писать универсальный код. увеличивается переносимость

3. Интерфейс на GWT, проверяющий, аутентифицирован ли пользователь. И, если нет, предлагающий ему аутентифицироваться путем ввода логина и пароля.

● Билет ?

1. Конвертер jsf

[Конвертер jsf](#) служат для преобразования данных в формат проперти, к которой они привязаны. По умолчанию есть конвертеры в числа и в дату. Можно написать свой конвертер, он должен реализовать интерфейс `converter`. конвертеры могут привязываться автоматически по типу данных, можно через вложенный тэг, можно через свойство

Что нужно для создания своего конвертера:

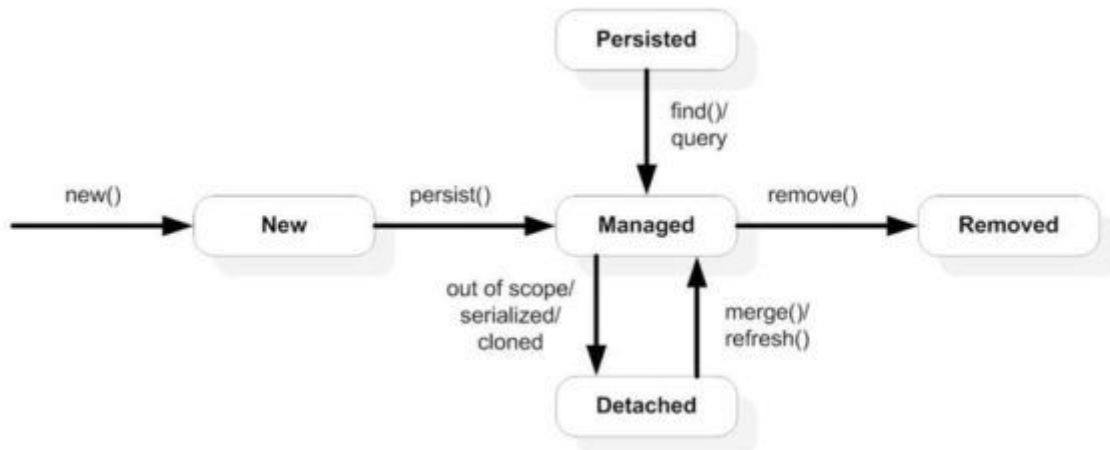
- Создать класс, реализующий интерфейс `Converter`
- Реализовать метод `getAsObject()`, для преобразования строкового значения поля в объект.
- Реализовать метод `getAsString`.
- Зарегистрировать конвертер в контексте Faces в файле `faces-config.xml`, используя элемент ИЛИ пометить аннотацией `@FacesConverter(name)`

файл `faces-config.xml` :

```
<converter>
  <converter-for-class>
    com.arcmind.contact.model.Group
```

```
</converter-for-class>
<converter-class>
    com.arcmind.contact.converter.GroupConverter
</converter-class>
</converter>
```

2. Интерфейс entitymanager и его методы.



EntityManager - базовый интерфейс для работы с хранимыми данными:

- Обеспечивает взаимодействие с Persistence Context.
- Можно получить через EntityManagerFactory.
- Обеспечивает базовые операции для работы с данными (CRUD).

Основные методы:

- 1) Для операций над Entity: persist ([добавление Entity под управление JPA, а именно PersistenceContext](#)), merge (обновление), remove (удаления), refresh (обновление данных), detach (удаление из управление JPA), lock (блокирование Entity от изменений в других thread),
- 2) Получение данных: find (поиск и получение Entity), createQuery, createNamedQuery, createNativeQuery, contains, createNamedStoredProcedureQuery, createStoredProcedureQuery
- 3) Получение других сущностей JPA: getTransaction, getEntityManagerFactory, getCriteriaBuilder, getMetamodel, getDelegate
- 4) Работа с EntityGraph: createEntityGraph, getEntityGraph
- 5) Общие операции над EntityManager или всеми Entities: close, isOpen, getProperties, setProperty, clear.

3. Виджет на gwt, для оповещений в vk, Twitter (вроде как-то так звучал)

● **Билет ?**

1. ajax в jsf

<f:ajax> можно указать листенера, который будет ловить евенты, можно указать, что отрендерить, можно указать, что отэкзакутить

2. Архитектура JPA

матрешка. использует ормпровайдер, который использует ждбц, который использует ждбц драйвер, который ходит в базу.

3. Интерфейс авторизации(логин+пароль) на React. На стороне сервера - REST API

```
import React from 'react';
import axios from "axios";

class LoginForm extends React.Component{

  constructor(props){
    super(props);
    this.state = {
      username: '',
      password: '',
      isAuth: false
    };
    this.onSubmit = this.onSubmit.bind(this);
  }

  onSubmit(e){
    e.preventDefault();
    var params = new URLSearchParams();
    params.append('username', this.state.username);
    params.append('password', this.state.password);
    axios.post('loginURL', params, {
      withCredentials: true
    }).then(
      response => {
        this.setState({isAuth: true});
      }
    ).catch(err => this.setState({isAuth: false}))
  }

  onChange(e){
    e.preventDefault();
    this.setState({[e.target.name]: e.target.value})
  }

  render() {
    return(
      <div>
        {
          isAuth ?
          <h2>authorised</h2>
          :
          <form onSubmit={this.onSubmit}>
            <input type="text" value={this.state.username}
              onChange={this.onChange.bind(this)} name="username" required/>
            <input type="password" value={this.state.password}
              onChange={this.onChange.bind(this)} name="password" required/>
            <button type="submit" className="ordinary" style={buttonBig}>Login</button>
          </form>
        }
      </div>
    )
  }
}
```

- **Билет ?**

1. Управляемые бины, конфигурация и что-то еще
2. JPA

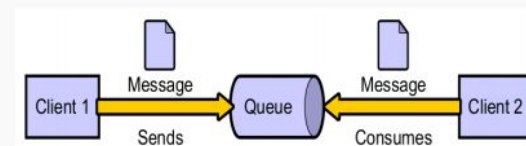
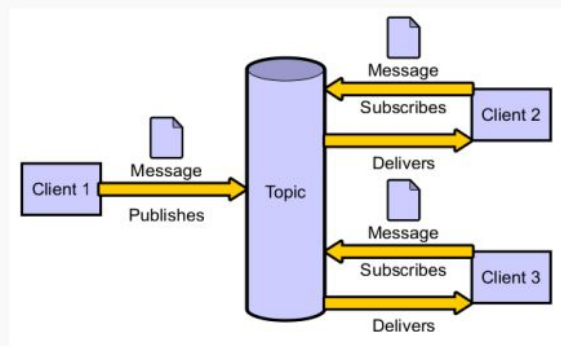
[jpa](#) это спецификация для [ORM-библиотек](#). а jdbc это стандарт. они нужны для работы с базами данных. jpa использует орм провайдер, который использует ждбс, который [использует ждбс драйвер](#) для доступа в базу данных. jpa позволяет абстрагироваться от особенностей жпа провайдера и писать универсальный код. увеличивается переносимость

3. Написать приложение на Vaadin, которое будет автоматически заполнять приказ об отчислении.

● Билет ?

1. JMS. Модели доставки сообщений

- Позволяет организовать асинхронный обмен сообщениями между компонентами.
- Две модели доставки сообщений — «подписка» (topic) и «очередь» (queue):



[Есть две модели: topic \(publisher / subscriber\) и queue \(point-to-point\)](#)

В случае топика поставщик - это что-то типа блоггера, он пишет сообщение, а подписота читает. Однако у сообщений есть тайм-аут, после которого их уже нельзя будет получить. В случае Queue сообщение исчезнет как только какой-либо потребитель его получит, то есть гарантируется, что только один юзер получит его. Таймаута на сообщении нет. Юзер может отказываться от сообщения, чтобы следующий в очереди мог посмотреть, не ему ли оно

2. GWT. Основные преимущества и недостатки

3. Реализуйте CRUD-интерфейс к таблице

Н ВИДЫ УЧЕБНОЙ ДЕЯТЕЛЬНОСТИ с помощью
Spring data

public interface repository extends CRUDRepository<Long,
ВИД> {} //интерфейс готов, к нему еще энити написать по
полям(наверное будет пример)

● **Билет ?**

1. Location Transparency, реализация в Java EE

[принцип прозрачного нахождения](#). мы можем писать код не думая о том, где находится объект, его будет вызывать контейнер. для реализации в java ee нужно реализовать стаб на стороне клиента и скелетон на стороне сервера. Контейнер будет таскать объекты с помощью gmi. Для клиента создаётся видимость целостности приложения, как будто оно не распределено по разным серверам с разными JVM, то есть для использования Remote EJB ему не надо делать дополнительных движений.

2. Rest в Spring. Spring RESTful

В спринг реализован rest путем RestController, все контроллеры, лежащие внутри класса, помеченного такой аннотацией будут автоматически сериализовать возвращаемые данные в JSON, так же можно указывать параметры в маппингах контроллеров, например @RequestMapping("/get/{entity}"), которые можно получать вместе с запросом. Под коробкой такого веб приложения может лежать Spring MVC или WebFlux. Также в Спринге есть REST клиент в различных реализациях RestTemplate, WebClient

3. Создать бин, конфигурируемый аннотациями, с именем myBean, контекст которого равен контексту другого бина - myOtherBean

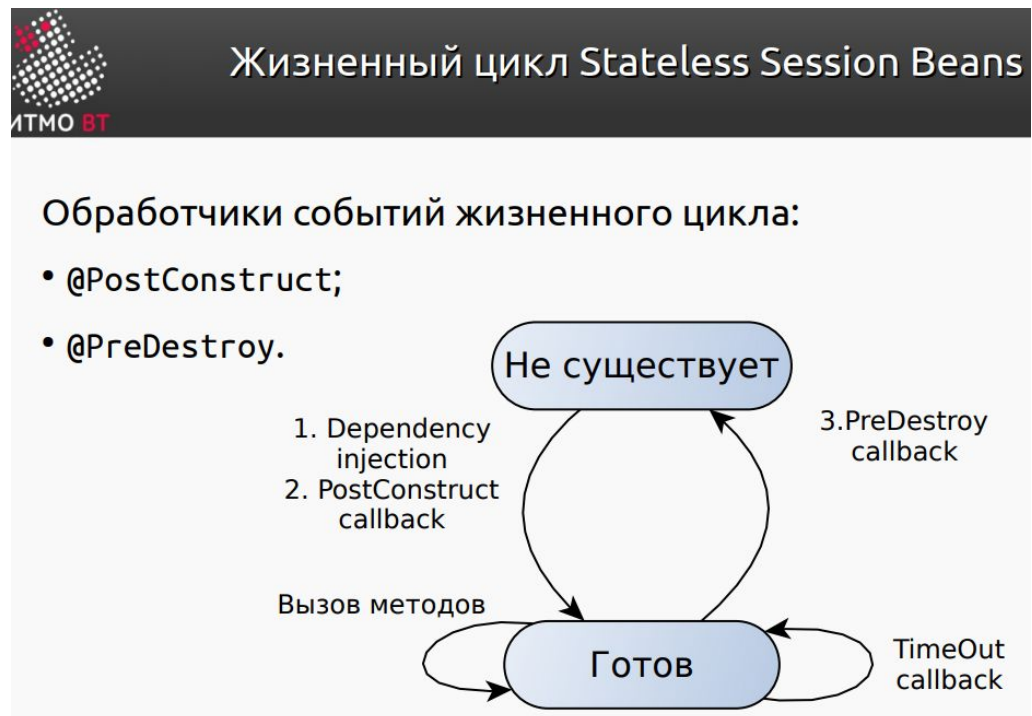
```
@ManagedBean(name="myBean")
class MyManagedBean {
    @ManagedProperty(value="#{myOtherBean}")
    private NeededBean neededBean;

    public NeededBean getNeededBean() {
        return neededBean;
    }

    public void setNeededBean(NeededBean neededBean) {
        this.neededBean = neededBean;
    }
}
```

● **Билет ?**

1. Жизненный цикл Session beans

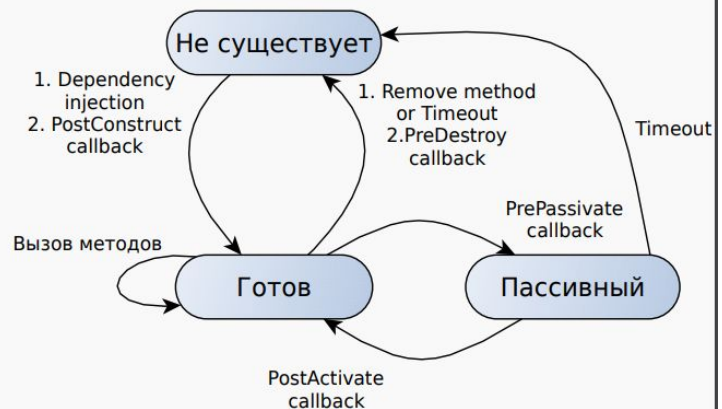




Жизненный цикл Stateful Session Beans

Обработчики событий жизненного цикла:

- @PostConstruct;
- @PreDestroy;
- @PostActivate;
- @PrePassivate.



В плюс к этому ещё есть пулы экземпляров бинов. Так как клиенты обращаются к проксям, реализующим бизнес-интерфейс, а с реальными объектами бинов дел не имеют, с их жизненным циклом можно вытворять разные штуки. Вот что происходит для каждого из трёх видов сессионных бинов:

- Stateful - у каждого клиента своя прокси, ведущая на свой и только свой экземпляр бина, потому что клиенту важно состояние его бина
- Stateless - у каждого клиента один и тот же интерфейс, который каким-то образом уводит пользователя к одному из экземпляров бина из пула
- Singleton - единый интерфейс и единый экземпляр заставят вас поразвлекаться с concurrency

2. Jsx, особенности, применение в реакт

JSX — синтаксис, похожий на XML / HTML, используемый в React, расширяет ECMAScript, так что XML / HTML-подобный текст может сосуществовать с кодом JavaScript / React. Синтаксис предназначен для использования препроцессорами (т. е. транспилерами, такими как Babel), чтобы преобразовать HTML-подобный текст, найденный в файлах JavaScript, в стандартные объекты JavaScript. Примеры(с и без):

```
function Button (props) {
  // Возвращает DOM элемент. Например:
  return <button type="submit">{props.label}</button>;}
// Отрисовываем компонент Button в браузере
ReactDOM.render(<Button label="Save" />, mountNode)

function Button (props) {
  return React.createElement(
    "button",
    { type: "submit" },
    props.label
  );
}
// Чтобы использовать Button вы должны написать что-то наподобие
// этого:
ReactDOM.render(
  React.createElement(Button, { label: "Save" }),
  mountNode
);
```

3. Написать правило навигации, для перехода с одной страницы на другую по нажатию кнопки

допустим кнопка генерит action main

```
<navigation-rule>
  <from-view-id>index.xhtml</from-view-id>
  <navigation-case>
    <from-outcome>main</from-outcome>
    <to-view-id>main.xhtml</to-view-id>
    <redirect/>
  </navigation-case>
</navigation-rule>
```

- **Билет ?**

1. Бизнес-логика, ejb, структура и классы. Локальные и удаленные что-то.

Пользователь хочет вызывать методы, но из-за того, что приложение может быть развёрнуто в разных JVM одновременно, контейнер должен иметь возможность управлять этими вызовами. Поэтому клиенту достаются не реальные объекты, а прокси, которые создаются контейнером и перенаправляют запросы куда надо. Чтобы контейнер знал, какие методы бина (класса, реализующего бизнес-логику, то есть Model из MVC) включать в прокси, нужен так называемый бизнес-интерфейс - интерфейс, содержащий объявления этих самых методов. Естественно, бин должен будет его реализовывать. EJB делятся на два вида (есть и другие деления, о них позже): Local и Remote. Когда бин помечается как локальный, мы договариваемся с контейнером, что данный бин будет вызываться только в пределах одной JVM, то есть его можно будет использовать по ссылке и ничего не придётся сериализовывать, что неплохо ускорит работу приложения. Раз уж на то пошло, то для локальных бинов даже необязательно делать бизнес-интерфейс, если пометить его аннотацией @LocalBean. А вот Remote бины имеют [все прелести RMI](#): передачу по значению а не по ссылке, а значит нужна сериализация + обязательны бизнес-интерфейсы.

Также EJB делятся на Session и Message-Driven. Вызов методов первых происходит напрямую, а вторых - по возникновению событий и только так (типа один большой бин-слушатель). MDB нас особо не интересуют. Session бины делятся на Stateful, Stateless и Singleton. Весьма очевидно из названий в чём их особенности, но если не слишком, то чекайте [ссылку](#).

2. SPA

одностраничное приложение. приложение, которое фактически содержит только одну html страницу и несколько разных представлений, подгружаемых динамически. если правильно сделать, то это экономит трафик, переносит часть нагрузки с серверов на клиента и спасает клиента от сбоев в сети. использует browser history api для переадресации и кнопки назад. может работать в оффлайн режиме

3. Что-то с БД, JPA Criteria api. Найти всех студентов со средним баллом ниже 4.0 и удалить

[Criteria API](#) - это когда вместо sql-подобных запросов в виде строки делается серия вызовов методов строителя запросов, определяющих новый запрос.

```
@Stateless
@LocalBean
public class StudentManagement {
    @PersistenceContext
    private EntityManager em;
    ...

    public void deleteStudents() {
        CriteriaBuilder cb = em.getCriteriaBuilder();

        CriteriaDelete<Student> delete =
        cb.createCriteriaDelete(Student.class);
        Root e = delete.from(Student.class);
        delete.where(cb.lessThan(e.get("avgGrade"), 4.0));
        this.em.createQuery(delete).executeUpdate();
    }
}
```

● Билет ?

1. stateless stateful singleton session bean. Сходство

ОТЛИЧИЯ ИТП

[Stateful ассоциируются с конкретным пользователем](#) + могут уходить в спячку (с возможностью умереть по достижении тайм-аута) чтобы ждать возвращения пользователя. Поэтому у каждого юзера своя прокси на свой ejb. Stateless могут менять обслуживаемых пользователей, поэтому в них не надо хранить состояние (хоть никто и не запретит). Благодаря такой особенности, нагрузку на сервера проще распределить между юзерами.

2. компоненты react. Что такое итп. Умные и глупые

компоненты

3. jsf поле многострочного ввода в которое можно ввести только строчные английские буквы

```
<h:inputTextArea><f:validateRegex pattern = "[a-z]*"/></h:inputTextArea>
```


Вопросы с se.ifmo.ru

Лаба №3

1. Технология JavaServer Faces. Особенности, отличия от сервлетов и JSP, преимущества и недостатки.

Структура JSF-приложения.

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

Преимущества JSF

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

Недостатки JSF

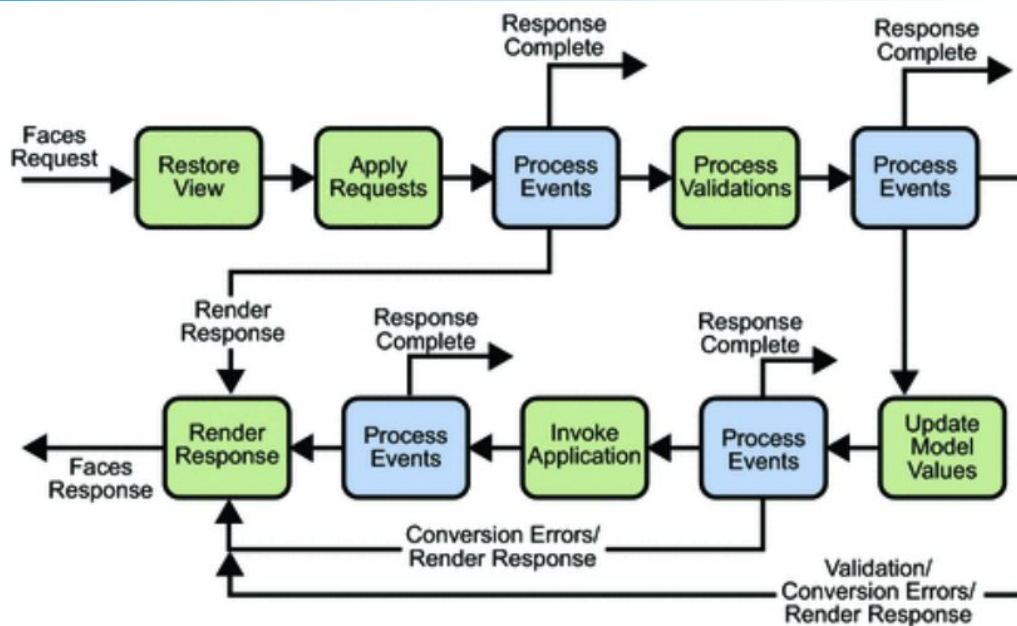
- Высокоуровневый фреймворк — сложно реализовывать не предусмотренную авторами функциональность.
- Сложности с обработкой GET-запросов (устранены в JSF 2.0).
- Сложность разработки собственных компонентов.

Структура JSF-приложения

- JSP-страницы с компонентами GUI

- Библиотека тегов
- Управляемые бины
- Доп. объекты(компоненты, конвертеры, валидаторы)
- Доп. теги
- Конфигурация – faces-config.xml
- Дескриптор развертывания – web.xml

Жизненный цикл обработки запросов



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

Facelets

■ Особенности

- Для создания страниц используется XHTML (transitional)
- Использование библиотек тегов (через пространства имен)
- Поддержка Expression Language

■ Преимущества

- Повторное использование кода (шаблоны и компоненты)
- Возможность настройки и корректировки работы компонентов
- Быстрая компиляция
- Проверка выражений EL на этапе компиляции
- Быстрый рендеринг компонентов

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 конверторами данных и многого 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 ограничивает JSP-страницей.

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

Особенности реализации JSF-компонент

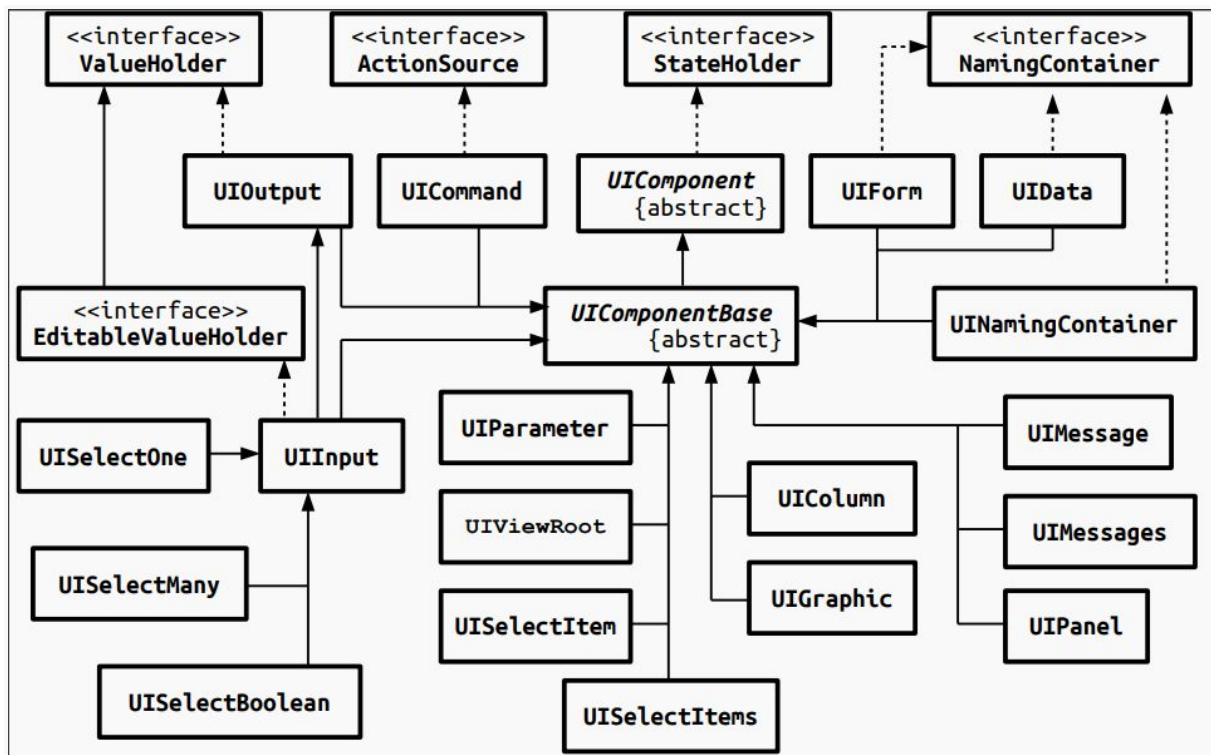
- Интерфейс строится из компонентов.
- Компоненты расположены на страницах JSP.
- Компоненты реализуют интерфейс `javax.faces.component.UIComponent`.
- Можно создавать собственные компоненты.
- Компоненты на странице объединены в древовидную структуру — представление.
- Корневым элементов представления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

Некоторые компоненты JSF: `<f:subview>`, `<h:selectOneMenu>`,

`<h:selectOneRadio>`, `<h:selectOneListbox>`, `<h:selectManyCheckbox>`,
`<selectManyListbox>`, `<selectManyMenu>`, `<h:textArea>`, ...

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

Иерархия классов (фрагмент)



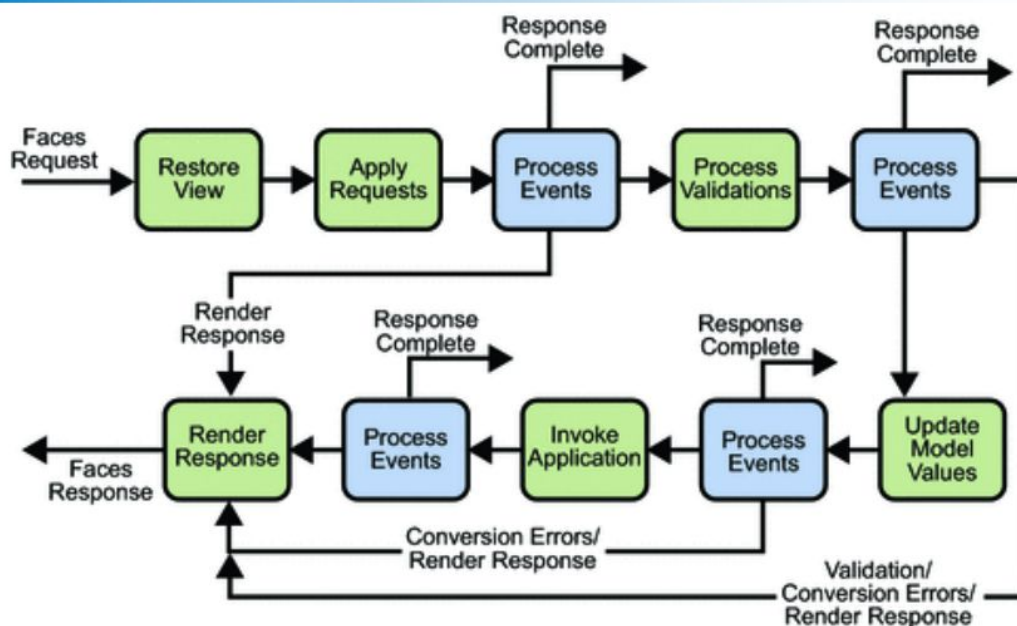
Дополнительные библиотеки компонентов.

[PrimeFaces](#), [RichFaces](#), [ICEFaces](#), [OpenFaces](#), [Trinidad](#), [Tomahawk](#).

Модель обработки событий

Жизненный цикл обработки запросов

- Значительно отличается от жизненного цикла в случае JSP
- Выражения EL вида `# { выражение }` (отложенное вычисление) имеют различный смысл на различных стадиях
 - На этапе формирования отклика позволяют вычислять значения, отправляемые в отклик
 - На этапе обработки запроса позволяют изменять состояние компонентов на сервере
- Операции жизненного цикла отличаются в случае первого и последующих запросов



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

- 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 значения 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.

4. Конвертеры и валидаторы данных.

JSF имеет встроенные конвертеры 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>
```

С пециализированные конвертеры

- Создать класс, реализующий интерфейс Converter
- Реализовать метод `getAsObject()`, для преобразования строкового значения поля в объект.
- Реализовать метод `getAsString`.
- Зарегистрировать конвертер в контексте Faces в файле `faces-config.xml`, используя элемент ИЛИ пометить аннотацией `@FacesConverter(name)`

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 валидации

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

```
public void validatePlayer(FacesContext context, UIComponent component,
    Object value) throws ValidatorException {
    // валидация
}
```

```
<h:selectOneRadio id="sportsPer" value="#{personView.per.likesTennis}"
    validator="#{personView.validatePlayer}" title="Play tennis"
    tabindex="6">
    <f:selectItem itemLabel="Yes" itemValue="Y" />
    <f:selectItem itemLabel="No" itemValue="N" />
</h:selectOneRadio>
```

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 реализован 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>
```

```
@FacesValidator("RValidator")
public class RValidator implements Validator {

    private static final double minR = 1;
    private static final double maxR = 4;

    public void validate(FacesContext facesContext, UIComponent uiComponent, Object o)
        throws ValidatorException {

        try {
            double r = Double.parseDouble(o.toString().replace(',', '.'));

            if (!(r >= minR && r <= maxR))
                throw new IllegalArgumentException();
        }
    }
}
```

```
    } catch (Exception e) {  
        FacesMessage msg =  
            new FacesMessage("R validation failed.",  
                "Неверный параметр R, пожалуйста, повторите ввод.");  
  
        msg.setSeverity(FacesMessage.SEVERITY_ERROR);  
        throw new ValidatorException(msg);  
    }  
}
```

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 `render` для прорисовки самого себя, согласно правилам в классе `Render`

`Renderer` - Отвечает за отображение компонента и преобразование ввода пользователя

`Validator, Converter`

`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.

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. Жизненным циклом 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.

Конфигурация управляемых бинов

`faces-config.xml`

```
<managed-bean>
  <managed-bean-name>customer</managed-bean-name>
  <managed-bean-class>CustomerBean</managed-bean-class>
  <managed-bean-scope>request</managed-bean-scope>
  <managed-property>
    <property-name>areaCode</property-name>
    <value>#{initParam.defaultAreaCode}</value>
  </managed-property>
</managed-bean>
```

С помощью аннотаций

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

Managed bean - бин, зарегистрированный в JSF, управляется JSF платформой. Managed 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 определяет 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 - экземпляр создаётся, но не привязывается ни к одной области жизни. Применяется когда к нему обращаются другие managed bean'ы, имеющие область жизни. *Бин без контекста.*

7. Конфигурация JSF-приложений. Файл faces-config.xml.

Класс FacesServlet.

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

faces-config.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<faces-config xmlns="http://java.sun.com/xml/ns/javaee"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://java.sun.com/xml/ns/javaee
    http://java.sun.com/xml/ns/javaee/web-facesconfig_1_2.xsd"
  version="1.2">
  <managed-bean>
    <managed-bean-name>calculator</managed-bean-name>

    <managed-bean-class>com.arcmind.jsfquickstart.model.Calculator</managed-bean-
class>
    <managed-bean-scope>request</managed-bean-scope>
  </managed-bean>
</faces-config>
```

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

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

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

Класс FacesServlet

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

8. Навигация в JSF-приложениях.

Механизм навигации 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>
```

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

Доступ осуществляется благодаря JDBC. Вместо него может быть использован какой-нибудь ORM-фреймворк (Hibernate, EclipseLink, ...), который всё равно будет использовать JDBC.

- JDBC — Java DataBase Connectivity
- JDBC API — высокоуровневый интерфейс для доступа к табличным данным, например к реляционной базе данных
- JDBC Driver API — низкоуровневый интерфейс для драйверов
- Пакеты java.sql (Core) и javax.sql (Extension)
- Стандарт взаимодействия с СУБД
- Для каждой СУБД используется свой драйвер

JDBC основан на концепции так называемых драйверов, позволяющих получать соединение с базой данных по специально описанному URL. Драйверы могут загружаться динамически (во время работы программы). Загрузившись, драйвер сам регистрирует себя и вызывается автоматически, когда программа требует URL, содержащий протокол, за который драйвер отвечает.

Драйвер для доступа к конкретной СУБД реализуется отдельно (зачастую разработчиками этой самой СУБД).

- Загрузка драйвера
 - ◊ `Class.forName()`
 - ◊ `jdbc.drivers=`
 - ◊ `META-INF/services/java.sql.Driver`
- Метод `getConnection()`
 - Возвращает `Connection`
 - `getConnection(String url)`
 - ◊ `URL = jdbc:protocol:name`
 - `getConnection(String url, Properties props)`
 - ◊ `Properties props = new Properties();`
 - ◊ `props.put("user", "s999999");`
 - ◊ `props.put("passwd", "xxx999");`
 - `getConnection(String url, String username, String passwd)`



Семейство интерфейсов Statement

- **Statement**
 - Статический SQL-запрос
 - `createStatement("SELECT * FROM table");`
- **PreparedStatement**
 - Динамический запрос с параметрами
 - `ps = prepareStatement("SELECT * FROM table WHERE id = ?");`
 - `ps.setInt(1, 15);`
- **CallableStatement**
 - Вызов хранимой процедуры
 - SQL: CREATE PROCEDURE
 - `cs = prepareCall("CALL get (?, ?)");`
 - `cs.setInt(1, 15);`
 - `cs.registerOutParameter(2, Types.VARCHAR);`



Методы execute...()

- **ResultSet** `executeQuery(String sql)`
 - для исполнения команды SELECT
 - Возвращает ResultSet
- **int** `executeUpdate(String sql)`
 - для выполнения запросов INSERT, UPDATE, DELETE
 - возвращает количество измененных строк
 - Для команд DDL возвращает 0
- **boolean** `execute(String sql)`
 - для выполнения любых запросов
 - Возвращает true, если результат — ResultSet
 - Возвращает false, если результат — updateCount

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

ORM (Object-Relational Mapping) - объектно-реляционное отображение — технология, которая позволяет осуществлять преобразование данных между ОО и реляционной моделью.

Существует три подхода к реализации ORM:

- Top-Down (Сверху-Вниз) – доменная модель приложения определяет реляционную.
- Bottom-up (Снизу-Вверх) – доменная модель строится на основании реляционной схемы.
- Meet-in-the-Middle – параллельная разработка доменной и реляционной моделей с учетом особенностей друг друга.

Сложности, возникающие при попытке отобразить один вид представления данных на другой, называют объектно-реляционным несоответствием.

Основные проявления объектно-реляционного несоответствия:

- Проблема идентичности.
- Представление наследования и полиморфизма.
- Проблема навигации между данными.


Решения первой проблемы:

Возможны три варианта:

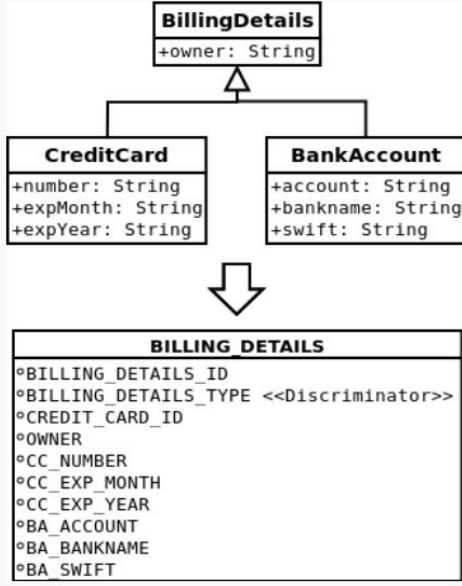
- 1) Обычный уровень хранения без управления идентичностью (no identity scope).
- 2) Уровень хранения с контекстно-управляемой (context-scoped) идентичностью .
- 3) Уровень хранения с жестким управлением идентичностью (process-scoped identity).

Решения проблемы наследования и полиморфизма (3 штуки):

- Одна общая таблица



Одна таблица для иерархии классов (Single Table Inheritance Pattern)



```

classDiagram
    class BillingDetails {
        +owner: String
    }
    class CreditCard {
        +number: String
        +expMonth: String
        +expYear: String
    }
    class BankAccount {
        +account: String
        +bankname: String
        +swift: String
    }
    BillingDetails <|-- CreditCard
    BillingDetails <|-- BankAccount
        
```

BILLING DETAILS
°BILLING_DETAILS_ID
°BILLING_DETAILS_TYPE <<Discriminator>>
°CREDIT_CARD_ID
°OWNER
°CC_NUMBER
°CC_EXP_MONTH
°CC_EXP_YEAR
°BA_ACCOUNT
°BA_BANKNAME
°BA_SWIFT

Все классы иерархии отображаются на одну таблицу базы данных.

Достоинства:

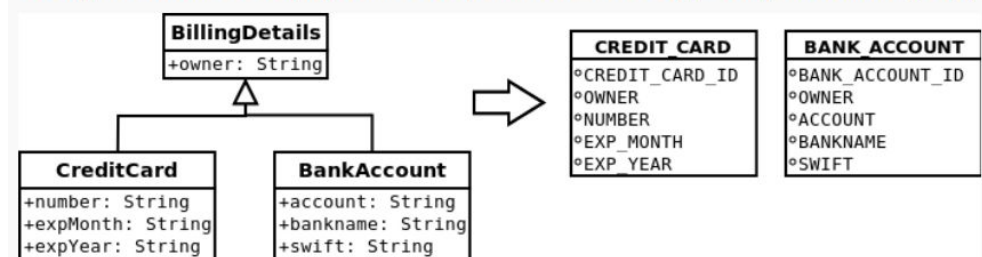
- 1) Наиболее простое решение.
- 2) Наиболее производительное решение.

Недостатки:

- 3) На поля подклассов нельзя накладывать null-ограничения.
- 4) Полученная таблица не нормализована.

- По таблице на каждого наследника

Каждый класс в иерархии отображается на отдельную таблицу БД.



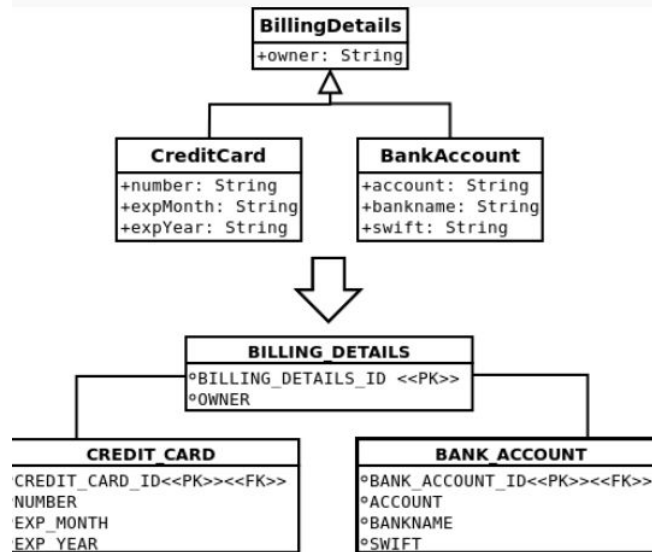
Достоинства:

Можно накладывать ограничения (not null) на поля подклассов.

Недостатки:

- 1) Полученные таблицы не нормализованы.
- 2) Плохая поддержка полиморфных запросов.
- 3) Низкая производительность.

- Таблицы и для предков, и для наследников



Каждый подкласс отображается на отдельную таблицу, которая содержит колонки, соответствующие только полям этого подкласса

Достоинства:

- 1) Таблицы нормализованы.
- 2) Лучший вариант для полиморфизма.
- 3) Существует возможность задавать ограничения на поля подклассов.

Недостатки:

Запросы выполняются медленнее, чем при использовании одной таблицы.

Примеры провайдеров ORM для Java: Hibernate, EclipseLink, TopLink

ORM - это, зачастую, надстройка над JDBC, повышающая уровень абстракции настолько, что программист уходит от использования пакета java.sql и не видит типичного JDBC-кода вообще.

JPA - часть Java EE - спецификация API, реализующая концепцию ORM.

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.

Помогает улучшить переносимость кода и в целом стандартизировать, а значит и упростить взаимодействие Java-приложений с СУБД. Основывается на понятии Entity - POJO-класс, виртуализирующий сущность в БД и описываемый с помощью аннотаций (@Entity, @Table, @Column, @Id, @OneToMany...) и/или с помощью xml.

Класс Entity

- Не должен быть внутренним (inner).
- Не должен быть final.
- Не должен иметь final методов.
- Должен иметь public-конструктор без аргументов.
- Атрибуты класса не должны быть public.

и должен содержать хотя-бы одно @Id-поле.

- Идентификатор (id) сущности, первичный ключ в БД.
- Уникально определяет сущность в памяти и БД.

	1. Simple id	
	<code>@Id int id;</code>	
Uses PK class	{	2. Compound id
		<code>@Id int id;</code>
		<code>@Id String name;</code>
	3. Embedded id	
		<code>@EmbeddedId EmployeePK id;</code>

- JPA поддерживает все стандартные виды связей:

- ▶ One-To-One
- ▶ One-To-Many
- ▶ Many-To-One
- ▶ Many-To-Many

- Поддерживаются однонаправленные и двунаправленные связи.

Для доступа к БД используется класс EntityManager.

```
► <T> T find(Class<T> entityClass, Object primaryKey)
► <T> T getReference(Class<T> entityClass, Object primaryKey)
► void persist(Object entity)
► <T> T merge(T entity)
► refresh(Object entity)
► remove(Object entity)
► void flush()
► void close();
```

Ещё там есть createQuery, createNativeQuery и прочее.

Для выполнения запросов используются SQL, JPQL (Java Persistence Query Language - SQL-подобный язык для работы с объектами вместо сущностей) и Criteria API (не SQL-подобные текстовые запросы, а методы)

11. Библиотеки ORM Hibernate и EclipseLink.

Особенности, API, сходства и отличия.

Hibernate - ORM-библиотека, ранее не реализовывавшая JPA, однако теперь всё ОК. Разве что не совсем всё так, как в спецификации, в отличие от EclipseLink - эталонной реализации JPA. Ничего критичного, правда, в этом несоблюдении спецификации нет.

В Hibernate есть своя версия JPQL - HQL.

Есть возможность интеграции с Apache Lucene для полнотекстового поиска по БД (Hibernate Search).

Существует мнение, что Hibernate надёжнее вследствие старости, в то время как EclipseLink быстрее.

Так как обе библиотеки реализуют JPA, их API максимально похож.

Быстрые и качественные основы гибера:

<https://docs.jboss.org/hibernate/orm/5.0/quickstart/html/>

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

JPA - часть Java EE - спецификация API, реализующая концепцию ORM.

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.

Помогает улучшить переносимость кода и в целом стандартизировать, а значит и упростить взаимодействие Java-приложений с СУБД. Основывается на понятии Entity - POJO-класс, виртуализирующий сущность в БД и описываемый с помощью аннотаций (@Entity, @Table, @Column, @Id, @OneToMany...) и/или с помощью persistence.xml.

Класс Entity

- Не должен быть внутренним (inner).
- Не должен быть final.
- Не должен иметь final методов.
- Должен иметь public-конструктор без аргументов.
- Атрибуты класса не должны быть public.

и должен содержать хотя-бы одно @Id-поле.

- Идентификатор (id) сущности, первичный ключ в БД.
- Уникально определяет сущность в памяти и БД.

```
1. Simple id
   @Id int id;

Uses { 2. Compound id
PK    @Id int id;
class { @Id String name;
       3. Embedded id
       @EmbeddedId EmployeePK id;
```

- JPA поддерживает все стандартные виды связей:
 - ▶ One-To-One
 - ▶ One-To-Many
 - ▶ Many-To-One
 - ▶ Many-To-Many
- Поддерживаются однонаправленные и двунаправленные связи.

Также для описания связей используется `@JoinColumn`, которая должна быть указана в классе-владельце связи, то есть того, который содержит в БД ссылку на вторую сущность.

Для доступа к БД используется класс `EntityManager`.

```
▶ <T> T find(Class<T> entityClass, Object primaryKey)
▶ <T> T getReference(Class<T> entityClass, Object primaryKey)
▶ void persist(Object entity)
▶ <T> T merge(T entity)
▶ refresh(Object entity)
▶ remove(Object entity)
▶ void flush()
▶ void close();
```

Ещё там есть `createQuery`, `createNativeQuery` и тому подобное.

Для выполнения запросов используются SQL, JPQL (Java Persistence Query Language - SQL-подобный язык для работы с объектами вместо сущностей) и Criteria API (не SQL-подобные текстовые запросы, а методы).

Persistence Context - это те сущности, с которыми идёт работа в приложении, то есть для них выполняется обмен данными с БД, для них гарантируется идентичность, с ними идёт работа в `EntityManager`'е.

Сущность бывает (согласно документации JBoss) в четырёх состояниях:

- New (transient) - сущность создана (используя `new`), но `PersistenceContext` о ней пока не знает.

- Managed - сущность, ассоциированная с контекстом с помощью метода persist в классе EntityManager.
- Detached - сущность, ранее бывшая ассоциированная с контекстом. Часто в таком состоянии находятся сущности, контекст которых был закрыт или которые были из него насильно удалены.
- Removed - сущность, которую было приказано удалить из БД.

В дополнение (там немного, но важно): <http://tomee.apache.org/jpa-concepts.html>

Лаба №4

1. Платформа Java EE. Спецификации и их реализации.

Java EE представляет из себя набор спецификаций и документации, описывающий архитектуру серверной платформы для задач средних и крупных предприятий.

Сервер приложений Java EE (часто называемый контейнером) — это реализация системы в соответствии со спецификацией, обеспечивающая работу модулей с логикой конкретного приложения.

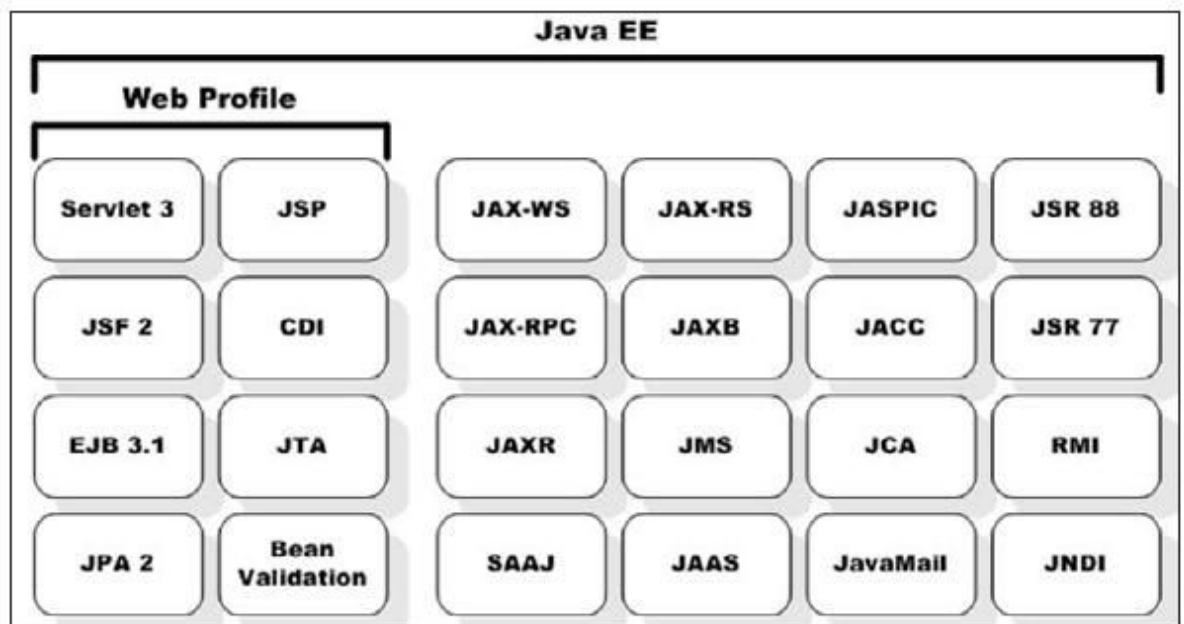


Figure 1: Java EE and the Web Profile

Деление на платформы (Java SE, Java EE, Java Card ...) появилось в Java EE 6 и позволяет сделать более «лёгкими» приложения, которым не нужен полный стек технологий Java EE. Существует только 2 профиля — Full и Web.

Сервер приложений может реализовывать спецификации не всей платформы, а конкретного профиля.

Пример Web profile: Tomcat

Пример Full profile: Glassfish

2. Принципы IoC, CDI и Location Transparency.

Компоненты и контейнеры.

Приложение состоит из компонентов и контейнера, управляющего жизненным циклом компонентов. Пример: Servlet - компонент, Glassfish - контейнер.

Есть такая штука - JNDI - набор API, позволяющий доставать объекты из контейнера по их именам. По сути, это замена обращению по ссылкам, и благодаря этому легко реализуется Dependency Lookup и Dependency Injection. Хранение объектов в таком случае можно представить как Map, где ключи - это имена, а значения - нужные объекты (например, ссылки на них).

Inversion of Control:

объекты создает не программист (используя `new`), а контейнер IoC. Применяется далеко не ко всем объектам в приложении, а только к управляемым (в Spring это классы с аннотациями `@Component`, `@Service` и т.д., в EJB — бобы `@Stateless`, `@Stateful`, `@MessageDriven`).

Контейнер не только создает объекты, но полностью управляет их жизненным циклом, вызывая на определенных этапах callback методы.

Dependency injection *(реализуется с помощью аннотаций):*

Вместо построения зависимостей в компоненте, где они нужны:

```
public class Something {  
    private Dependency dependency = new Dependency()  
}
```

Мы принимаем их извне, что избавляет компонент от необходимости управлять ими:

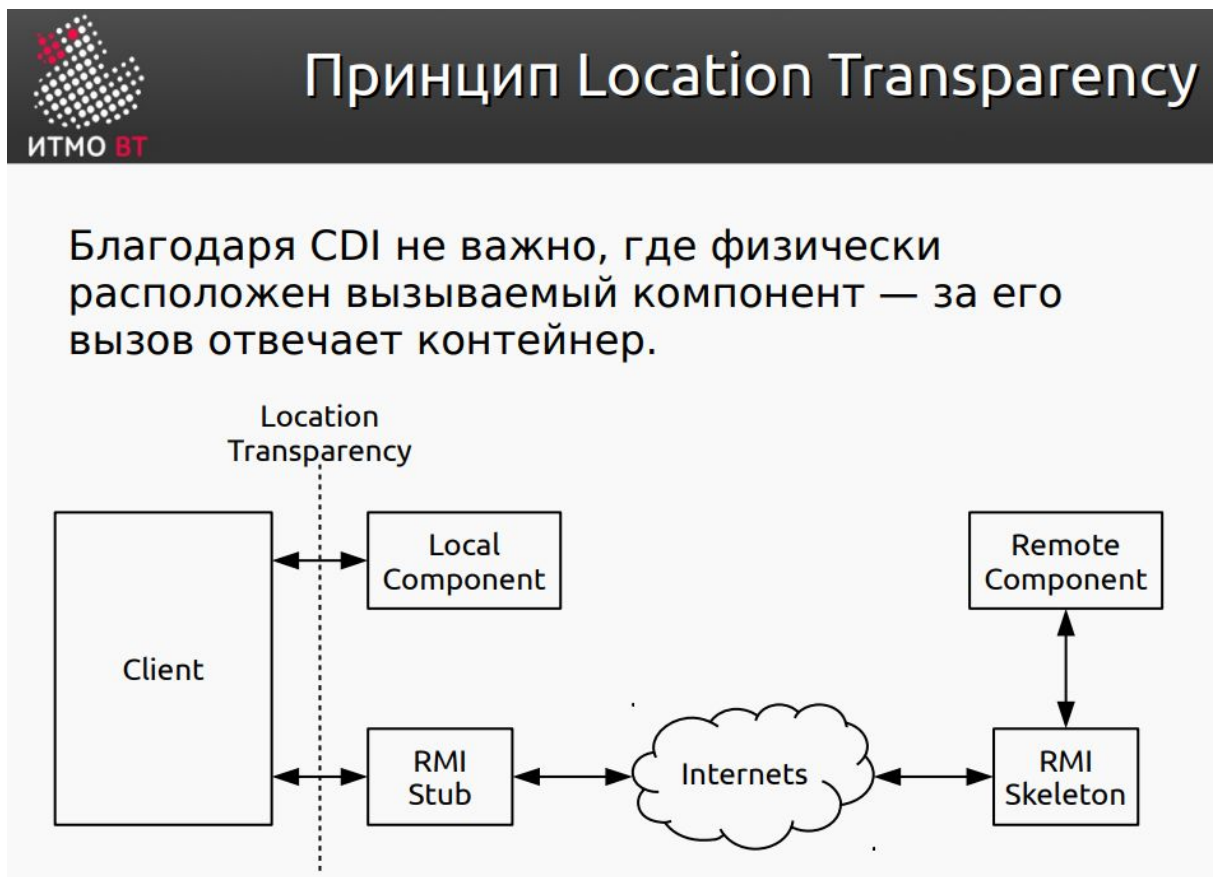
```
public class Something {  
    private Dependency dependency;  
  
    public Something (Dependency dependency) {  
        this.dependency = dependency;  
    }  
}
```

Чтобы нам не пришлось вручную собирать все зависимости для создания компонентов, контейнер IoC предоставляет возможность внедрения инстанций в поля, помеченные специальной аннотацией:

```
public class SomethingDI {  
    @Autowired private Dependency dependency; // Spring  
    @EJB private EjbDependency ejbDependency; // EJB  
}
```

Location Transparency:

Клиент имеет прокси, обращается с ней как с нужным объектом. Прокси перенаправляет его вызовы куда следует: если нужный компонент находится локально, идёт к нему обращение по ссылке; если удалённо (в другой JVM) - необходима сериализация передаваемых данных.



3. Управление жизненным циклом компонентов.

Дескрипторы развёртывания.

По большей части, жизненным циклом компонентов управляет контейнер. Тем не менее, есть возможность влиять на этот процесс с помощью аннотаций / xml / прочих файлов настроек. Дескрипторы - это как раз такие файлы (в том числе xml). В них можно задавать свойства используемых в приложении компонентов: имена, ссылки, параметры. Также можно настраивать то, как итоговое приложение будет собираться и как оно будет взаимодействовать с внешним миром.

Дескриптор развёртывания - конфигурационный файл артефакта. Дескриптор развёртывания описывает то, как компонент, модуль или приложение (такое, как веб-приложение или приложение предприятия) должно быть развёрнуто

Примеры дескрипторов развёртывания:

web.xml - дескриптор развёртывания веб-приложений (упаковываемых обычно в .war архивы)

ejb-jar.xml - дескриптор развёртывания EJB-приложения

application.xml - дескриптор развёртывания приложения, использующего несколько web.xml / ejb-jar.xml

4. Java EE API. Виды компонентов. Профили платформы Java EE.

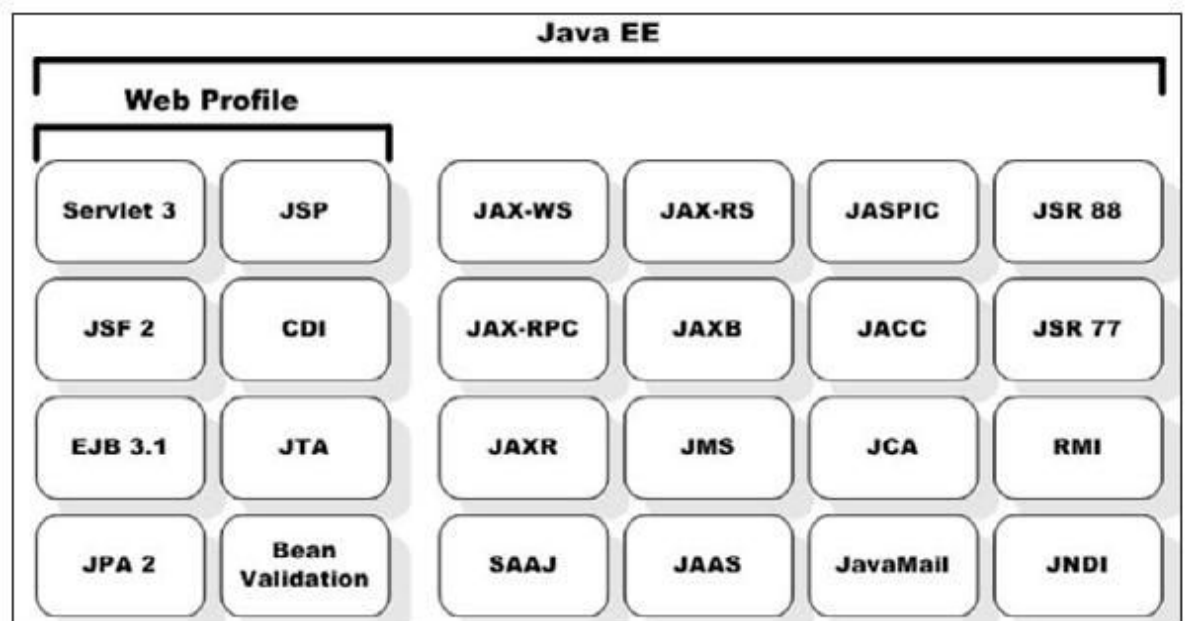


Figure 1: Java EE and the Web Profile

Деление на платформы (Java SE, Java EE, Java Card ...) появилось в Java EE 6 и позволяет сделать более «лёгкими» приложения, которым не нужен полный стек технологий Java EE. Существует только 2 профиля — Full и Web. Сервер приложений может реализовывать спецификации не всей платформы, а конкретного профиля.

Пример Web profile: Tomcat

Пример Full profile: Glassfish

Виды компонентов:

- веб-компоненты (сервлеты, jsp, ...) - формируют содержимое представления
- Java-бины (POJO, ManagedBean, EJB) - позволяют изменять данные и осуществлять их временное хранение, взаимодействовать с базами данных и веб-службами, а также отображать содержимое в ответ на запросы клиентов.

5. Компоненты EJB. Stateless & Stateful Session Beans.

EJB Lite и EJB Full.

EJB (Enterprise Java Bean) — спецификация для разработки серверных компонентов, реализующих бизнес-логику.

Компоненты — бобы, которые делятся на *session beans* (заседательные бобы) и *message driven beans* (бобы, движимые посланиями).

Session beans в свою очередь делятся на:

- *stateful*: у каждого клиента своя инстанция, в которой хранится его состояние
- *stateless*: одна и та же инстанция обеспечивает запросы нескольких клиентов => лучше масштабируются, но не могут сохранять состояние между последовательными обращениями клиента
- *singleton*: одна инстанция на все приложение => общее для всех клиентов состояние

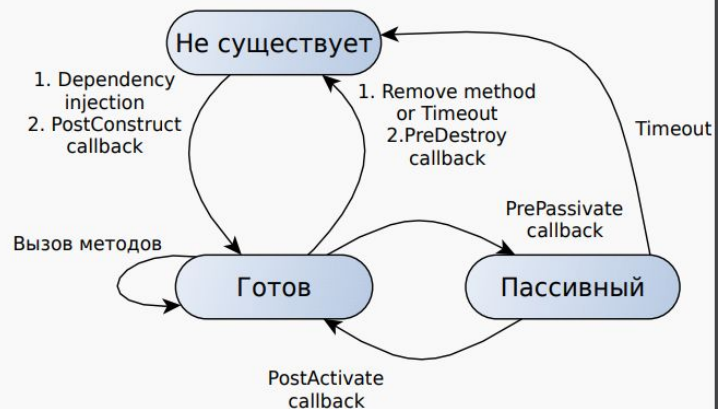
Немного про жизненные циклы:



Жизненный цикл Stateful Session Beans

Обработчики событий жизненного цикла:

- `@PostConstruct`;
- `@PreDestroy`;
- `@PostActivate`;
- `@PrePassivate`.



В плюс к этому ещё есть пулы экземпляров бинов. Так как клиенты обращаются к проксям, реализующим бизнес-интерфейс, а с реальными объектами бинов дел не имеют, с их жизненным циклом можно вытворять разные штуки. Вот что происходит для каждого из трёх видов сессионных бинов:

- **Stateful** - у каждого клиента своя прокси, ведущая на свой и только свой экземпляр бина, потому что клиенту важно состояние его бина
- **Stateless** - у каждого клиента один и тот же интерфейс, который каким-то образом уводит пользователя к одному из экземпляров бина из пула
- **Singleton** - единый интерфейс и единый экземпляр заставят вас поразвлекаться с concurrency

Message driven beans выполняют метод в ответ на получение сообщения из определенной очереди JMS.

Аннотация `@EJB` служит для CDI и предоставляет клиентам не сам объект, а прокси, через который можно получить доступ к методам бизнес-интерфейсов. Может осуществляться доступ как к локальным, так и удаленным (в другой JVM) объектам.

6. Работа с электронной почтой в Java EE. JavaMail API.

JavaMail API - фреймворк для работы с электронной почтой в Java с поддержкой протоколов SMTP, MIME, POP и некоторые другие.

Предоставляется следующее API из пакета javax.mail:

- Session - класс, открывающий дверь в мир JavaMail
- Message - абстрактный класс (используются его наследники, например MimeMessage)
- Address - абстрактный класс
- Authenticator - абстрактный класс для защиты сообщений на сервере
- Transport
- Store
- Folder

Пример кода:

```
// Recipient's email ID needs to be mentioned.
String to = "destinationemail@gmail.com";

// Sender's email ID needs to be mentioned
String from = "fromemail@gmail.com";
final String username = "manishaspatil";//change accordingly
final String password = "*****";//change accordingly

// Assuming you are sending email through relay.jangosmtp.net
String host = "relay.jangosmtp.net";

Properties props = new Properties();
props.put("mail.smtp.auth", "true");
props.put("mail.smtp.starttls.enable", "true");
props.put("mail.smtp.host", host);
props.put("mail.smtp.port", "25");

// Get the Session object.
Session session = Session.getInstance(props,
    new javax.mail.Authenticator() {
        protected PasswordAuthentication getPasswordAuthentication() {
            return new PasswordAuthentication(username, password);
        }
    });

try {
    // Create a default MimeMessage object.
    Message message = new MimeMessage(session);

    // Set From: header field of the header.
    message.setFrom(new InternetAddress(from));

    // Set To: header field of the header.
    message.setRecipients(Message.RecipientType.TO,
        InternetAddress.parse(to));
```

```
// Set Subject: header field
message.setSubject("Testing Subject");

// Now set the actual message
message.setText("Hello, this is sample for to check send " +
    "email using JavaMailAPI ");

// Send message
Transport.send(message);

System.out.println("Sent message successfully....");

} catch (MessagingException e) {
    throw new RuntimeException(e);
}
```

7. JMS. Реализация очередей сообщений. Способы доставки сообщений до клиента. Message-Driven Beans.

Java Message Service — стандарт для асинхронного распределенного взаимодействия программных компонентов (которые могут находиться на одном компьютере, в одной локальной сети, или быть связаны через Интернет) путем рассылки сообщений.

JMS поддерживает две модели коммуникации: *point-to-point* и *publish-subscribe (pubsub)*.

В *point-to-point* сообщения от разных отправителей адресуются определенной очереди, к которой подключаются клиенты. При этом для каждого сообщения гарантируется, что оно будет доставлено одному и только одному клиенту.

В *pubsub* сообщения адресуются определенному topic'у, на которые подписываются клиенты. Каждое сообщение может быть получено несколькими клиентами или не получено вообще, если подписчиков на момент доставки не было.

Существует несколько реализаций JMS провайдеров (RabbitMQ, Open Message Queue, ...)

Про Message-Driven заданий не будет

8. Понятие транзакции. Управление транзакциями в Java EE. JTA.

Транзакция — группа последовательных операций, представляет собой логическую единицу работы с данными. Транзакция либо выполняется успешно и целиком, соблюдая целостность данных, либо не производит никакого эффекта на данные.

Java Transaction API позволяет выполнять распределенные транзакции, т.е. транзакции, читающие и обновляющие данные на разных сетевых ресурсах (которыми могут быть различные серверы баз данных, JMS).


JTA предоставляет высокоуровневый интерфейс для управления транзакциями (`begin`, `commit`, `rollback`), избавляя от необходимости работы с каждым ресурсом по-своему (интерфейс транзакций в JDBC, например, немного отличается от интерфейса JMS).

Транзакция координируется `transaction manager`'ом. Взаимодействие с ресурсами осуществляется через `resource manager`'ы.

Транзакции могут быть объявлены:

- декларативно — аннотацией `@Transactional` на отдельном методе или всем классе, при этом `rollback` происходит при необработанном `RuntimeException`
 - программно — вызывая `begin`, `rollback`, `commit` у [UserTransaction](#)
- В дополнение (там немного, но важно): <http://tomee.apache.org/jpa-concepts.html>

9. Веб-сервисы. Технологии JAX-RS и JAX-WS.

 **Веб-сервисы**

- Позволяют организовать взаимодействие между сетевыми ресурсами по стандартизированному протоколу.
- Ресурсы могут работать на любой платформе.
- Данные «упаковываются» в XML и передаются по HTTP.
- Основные стандарты — SOAP, WSDL и WS-I.
- На платформе Java EE реализуются JAX-WS API и JAX-RS API.
- Основа SOA (Service Oriented Architecture).

Пример JAX-WS (RPC веб-сервис)


```
package example;

import javax.jws.*;

@WebService
public class SayHello {

    @WebMethod
    public String getGreeting(String name){
        return "Hello " + name;
    }

}
```

 **Пример реализации клиента веб-сервиса (JAX-WS)**

```
import javax.xml.ws.WebServiceRef;

public class WSTest {

    public WSTest() { }

    public static void main(String[] args) {
        SayHelloService service = new SayHelloService();
        SayHello port = service.getSayHelloPort();
        System.out.println(port.sayHello("Duke"));
    }

}
```

Содержание

Пример JAX-RS (REST веб-сервис)

```
package com.example;

import javax.enterprise.ApplicationScoped;
import javax.ws.rs.*;

@ApplicationScoped
@Path("/")
public class Rest {
    @GET
    @Path("echo")
    public String echo(@QueryParam("q") String original) {
        return original;
    }
}
```

10. Платформа Spring. Сходства и отличия с Java EE.
11. Модули Spring. Архитектура Spring Runtime. Spring Security и Spring Data.
12. Реализация IoC и CDI в Spring. Сходства и отличия с Java EE.
13. Реализация REST API в Java EE и Spring.
14. React JS. Архитектура и основные принципы разработки приложений.

15. Компоненты React. State & props. "Умные" и "глупые" компоненты.
16. Разметка страниц в React-приложениях. JSX.
17. Навигация в React-приложениях. ReactRouter.
18. Управление состоянием интерфейса. Redux.
19. Angular: архитектура и основные принципы разработки приложений.
20. Angular: модули, компоненты, сервисы и DI.
21. Angular: шаблоны страниц, жизненный цикл компонентов, подключение CSS.
22. Angular: клиент-серверное взаимодействие, создание, отправка и валидация данных форм.