

## 1 Билет

### 1) Что такое JSF

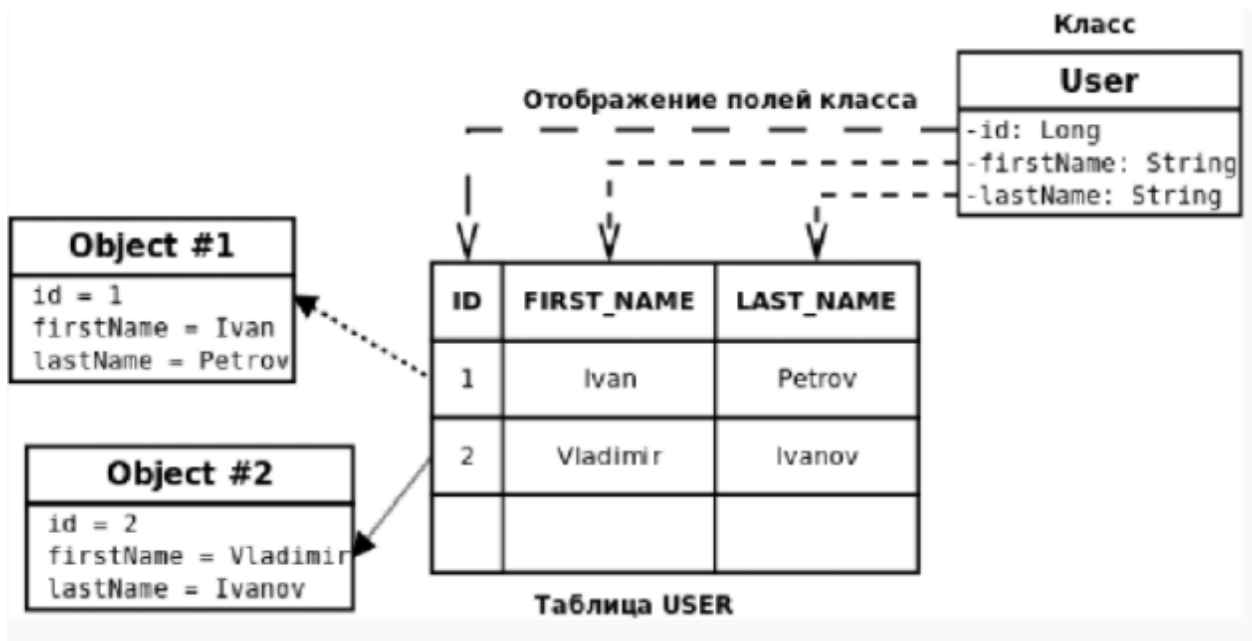
**JavaServer Faces (JSF)** — это фреймворк для веб-приложений, написанный на Java. Он служит для того, чтобы облегчать разработку пользовательских интерфейсов для Java EE приложений. В отличие от прочих MVC фреймворков, которые управляются запросами, подход JSF основывается на использовании компонентов.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

Структура JSF-приложения:

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

## 2) Что такое ORM

**ORM** (Object-Relational Mapping) - технология программирования, которая позволяет связать базы данных с объектами языков программирования, создавая “виртуальную объектную базу данных”



## Подходы к реализации ORM

Существует три подхода:

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

### 3) Как реализовать на ангуляре страницу для вывода соцсети

```
export interface SocialMedia {  
  link:string,  
  user:string,  
  company?:boolean  
}
```

создаем html файл нашего компонента, и прописываем элементы как будет выглядеть наш вывод вывод соц сети

```
import { Component, Input } from '@angular/core';  
import { SocialMedia } from './social-media';  
  
@Component({  
  selector: 'ces-social-media',  
  templateUrl: './social-media.component.html',  
  styleUrls: ['./social-media.component.scss']  
})  
export class SocialMediaComponent {  
  @Input() facebook:SocialMedia;  
  @Input() twitter:SocialMedia;  
  @Input() linkedIn:SocialMedia;  
  @Input() medium:SocialMedia;  
  @Input() vertical:boolean = true;  
  
  facebookLink:string = "https://www.facebook.com/";  
  twitterLink:string = "https://twitter.com/";  
  linkedInLink:string = "https://www.linkedin.com/";  
  linkedInLinkCompany:string = "https://www.linkedin.com/company/"  
  mediumLink:string = "https://medium.com/@";
```

Вставляем например в footer нашей страницы

```
<ces-social-media ngClass="social-media-footer" [facebook]="{link:  
'egor_redgry', user: 'redgry' }" />
```

<https://levelup.gitconnected.com/how-to-create-a-social-media-component-in-angular-for-all-of-your-users-549fa945ac4a>

## 2 Билет

### 1) Структура JSF

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

#### Структура JSF-приложения:

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

### 2) Наследование и полиморфизм в ORM

При объектно-реляционном отображении наследование и полиморфизм тесно связаны.

#### Три способа реализации наследования:

##### 1. Одна таблица для всех классов.

Плюсы: простота и производительность

Минусы: отсутствие null ограничений, не нормализованная

таблица

##### 2. Своя таблица на каждый класс

Плюсы: возможность null ограничений

Минусы: плохая поддержка полиморфных записей, не

нормализованная таблица

##### 3. Своя таблица на каждый подкласс

Плюсы: нормализованная таблица, возможность null ограничений

Минусы: низкая производительность

### 3) EJB калькулятор на 4 операции

```
1. @Remote
2. public interface CalculatorInterface{
3.
4.     public double add(double a, double b);
5.     public double min(double a, double b);
6.     public double mult(double a, double b);
7.     public double div(double a, double b);
8.
9. }
10.
11.
12. @Stateless(name="calculator")
13. public class Calculator implements CalculatorInterface {
14.     public double add(double a, double b){
15.         return a+b;
16.     }
17.     public double min(double a, double b){
18.         return a-b;
19.     }
20.     public double mult(double a, double b){
21.         return a*b;
22.     }
23.     public double div(double a, double b){
24.         return a/b;
25.     }
26. }
```

## 3 Билет

### 1) Конвертер JSF

Конвертеры используются для преобразование данных компонента в заданный формат. Существуют стандартные конвертеры для основных типов данных. BigDecimal BigInteger Boolean Byte Character DateTime Double Float.

Для создания Кастомного конвертера необходимо:

- 1) Создать класс, реализующий интерфейс Converter
- 2) Реализовать метод `getAsObject()`, который будет вызываться для преобразования строкового значения поля в объект.
- 3) Реализовать метод `getAsString()`, который будет вызываться для получения строкового представления объекта.
- 4) Зарегистрировать конвертер в файле `faces-config.xml`, используя элемент `<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)

persist() - добавляет новый экземпляр Entity в БД, сохраняет состояние Entity и относящихся к ней ссылок.

find() - получает управляемый экземпляр Entity (по идентификатору) - возвращает null, если заданный объект не найден.

remove() - удаляет управляемую Entity. Опционально производит каскадное удаление отмеченных объектов.

merge() - создается управляемая копия переданной отсоединенной Entity. Поддерживает каскадное распространение.

### 3) Сервлет, реализующий CRUD

Создаем класс и указываем аннотацию с ссылкой на данный сервлет

@WebServlet

Переопределяем метод, который будет получать у нас вид запроса, для

@doGet - будет Read, для @doPost - будет Create, для @doDelete - будет

Delete, для @doPut - будет обновление, прописываем нашу логику внутри

(картинки для примера) и отправляем ответ клиенту с помощью указания в

response наш sendRedirect и URL (ссылки)

```
public Customer createCustomer(int id, String name) {  
    Customer cust = new Customer(id, name);  
    entityManager.persist(cust);  
    return cust;  
}
```

```
public void removeCustomer(Long custId) {  
    Customer cust =  
        entityManager.find(Customer.class, custId);  
    entityManager.remove(cust);  
}
```

```
public Customer storeUpdatedCustomer(Customer cust)  
    return entityManager.merge(cust);  
}
```

## 4 Билет

### 1) FacesServlet. Конфигурация

**FacesServlet** - создает 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.

**Например:** Мы отправляем форму, которая с помощью FacesServlet связывает со слушателем, выполняет действия и отправляет ответ и FacesServlet понимает куда нужно его отправить с помощью конфигурации.

**Конфигурация** задается в дескрипторе web.xml

Мы там указываем сервлет (его название и его класс), потом мы можем указать куда должен мапиться наш сервлет (указываем название сервлета и URL)

```
1 <web-app xmlns="http://java.sun.com/xml/ns/javaee" version="2.5">
2   <servlet>
3     <servlet-name>comingsoon</servlet-name>
4     <servlet-class>mysite.server.ComingSoonServlet</servlet-class>
5   </servlet>
6   <servlet-mapping>
7     <servlet-name>comingsoon</servlet-name>
8     <url-pattern>/*</url-pattern>
9   </servlet-mapping>
10 </web-app>
```

### 2) Hibernate. Задание сущностей, типы соединений, типы языков

**Hibernate** - ORM-фреймворк один из основных реализаций спецификации JPA (API). **Сущности задаются с помощью аннотации @Entity** или через persistence.xml, где прописывается различное описание сущности, к какой таблице относятся, столбы и т.д. (также у сущности должно быть поле с аннотацией @Id, говорящее Hibernate, что поле используется как первичный ключ в БД, чтобы отличать объекты друг от друга)

**4 вида соединений:** OneToOne, OneToMany, ManyToOne, ManyToMany. Hibernate - генерирует SQL вызовы и освобождает от ручной обработки результирующего набора данных и конвертации объектов, сохраняя приложение портируемым во все SQL базы данных.

**Языки SQL и JDBC**



### 3) Redux Storage, описывающее состояние CRUD интерфейса к списку студентов

```
import {
  CREATE,
  READ,
  UPDATE,
  DELETE } from 'constants'

const initialState = {
  //массив студентов
  students = []
}

export function studentsReducer(state = initialState, action){
  switch(action.type){

    case CREATE:
      return {...state, state.students: [...state.students, action.payload]}
    case UPDATE:
      return {...state, students[action.payload.index]: action.payload}
    case DELETE:
      return {...state, state.students: [
        ...state.students.slice(0, action.index),
        ...state.students.slice(action.index + 1)
      ]}
    case READ:
      return {...state, readen: "ну прочитай, браток"}
  }
}
```

## 5 Билет

### 1) Контекст управляемых бинов. Конфигурация контекста.

Конфигурация контекста задается через faces-config.xml или с помощью аннотаций 6 видов (внедрение аннотации выполняется до внедрения XML. Таким образом конфигурации XML будут иметь приоритет над аннотациями):

@NoneScoped - контекст не определен, жизненным циклом управляют другие бины.

@RequestScoped (по умолчанию) - контекст-запроса

@ViewScoped - контекст-страницы

@SessionScoped - контекст-сессии

@ApplicationScoped - контекст-приложения

@CustomScoped - бин сохраняется в Map, программист сам управляет его жизненным циклом

### 2) Связи между сущностями в JPA

С помощью аннотации @Entity мы указываем что данный класс является сущностью отображением в базе данных. Связь между сущностями - это зависимость одной сущности от другой:

Виды:

OneToOne

OneToMany

ManyToOne

ManyToMany

Двунаправленные - классы хранят ссылки друг на друга

Однонаправленные - ссылка хранится только в одном из классов

### 3) Страница на Angular проверяющая по наличию куки и выдает либо форму входа если куки нет, либо что-то другое если кука есть.

(Создаем компонент, который отвечает за вывод приложения, а другой за форму, с помощью директивы \*ngIf мы проверяем на наличие нужной нам куки, если она есть то мы выводим компонент нашего приложения, если её нет, тогда выводим форму входа. В компонентах мы указываем с помощью аннотации @Component наш селектор, шаблон URL и ссылку на стили. В самом классе мы уже указываем нужные нам методы, объекты, конструкторы, которые выполняют свои задачи)

```
import { Component } from '@angular/core';
import { FormBuilder } from '@angular/forms';

import { CartService } from '../cart.service';

@Component({
  selector: 'app-cart',
  templateUrl: './cart.component.html',
  styleUrls: ['./cart.component.css']
})
export class CartComponent {
  items = this.cartService.getItems();
  checkoutForm = this.formBuilder.group({
    name: '',
    address: ''
  });
  constructor(
    private cartService: CartService,
    private formBuilder: FormBuilder,
  ) {}

  onSubmit(): void {
    // Process checkout data here
    this.items = this.cartService.clearCart();
    console.warn('Your order has been submitted', this.checkoutForm.value);
    this.checkoutForm.reset();
  }
}
```

4) React. Строка с автодополнением, слова для него нужно забрать с API.


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

export class Autocomplete extends React.Component{
  constructor(props){
    super(props)
    this.state = {
      list: this.getList()
    }
  }

  getList(){
    axios.get('/someURL')
      .then(result => {
        return result
      })
  }

  render() {
    return(
      <div class={"AutoComplete"}>
        <input list={"options"}/>
        <datalist id={"options"}>
          {this.state.list.map((key, item)=>
            <option key={key} value={item}/>
          )}
        </datalist>
      </div>
    )
  }
}
```

1) CDI vs IoC в Java EE



## Принципы IoC и CDI

- IoC (применительно к Java EE):
  - Жизненным циклом компонента управляет контейнер (а не программист).
  - За взаимодействие между компонентами отвечает тоже контейнер.
- CDI — позволяет снизить (или совсем убрать) зависимость компонента от контейнера:
  - Не требуется реализации каких-либо интерфейсов.
  - Не нужны прямые вызовы API.
  - Реализуется через аннотации.

**IoC** — один из принципов, приближающий наш код к слабой связанности.

**IoC** — это делегирование части наших обязанностей внешнему компоненту. Dependency Injection реализация IoC.

**CDI** - является частью веб-профиля Java EE 6 и основано на внедрении зависимостей для Java (@Inject, @Named )

**CDI** добавляет различные компоненты EE, такие как @RequestScoped, перехватчики / декораторы, производители, события и основу для интеграции с JSF, EJB и т. д. Компоненты Java EE, такие как EJB, были переопределены для построения поверх CDI (= > @Stateless теперь является управляемым компонентом CDI с дополнительными службами).

Ключевой частью CDI, помимо его возможностей DI, является знание контекстов bean компонентов и управление жизненным циклом bean-компонентов и зависимостями в этих контекстах (таких как @RequestScoped или @ConversationScoped).

## 2) Spring Data с JPA

**Spring Data** — дополнительный удобный механизм для взаимодействия с сущностями базы данных, организации их в репозитории, извлечение данных, изменение, в каких то случаях для этого будет достаточно объявить интерфейс и метод в нем, без имплементации.

Основное понятие в Spring Data — это репозиторий. Это несколько интерфейсов которые используют JPA Entity для взаимодействия с ней. Так например интерфейс

```
public interface CrudRepository<T, ID extends Serializable> extends  
Repository<T, ID>
```

обеспечивает основные операции по поиску, сохранения, удалению данных (CRUD операции)

Методы запросов из имени метода:

Запросы к сущности можно строить прямо из имени метода. Для этого используется механизм префиксов find...By, read...By, query...By, count...By, и get...By, далее от префикса метода начинается разбор остальной части.

Вводное предложение может содержать дополнительные выражения, например, Distinct. Далее первый By действует как разделитель, чтобы указать начало фактических критериев. Можно определить условия для свойств сущностей и объединить их с помощью And и Or.

## 3) Написать часть кода JSF, которая будет создавать компонент, который будет проводиться по списку новостей и выводить поля объекта.

```
<%@taglib prefix="c" uri="http://java.sun.com/jsp/jstl/core"%>
```

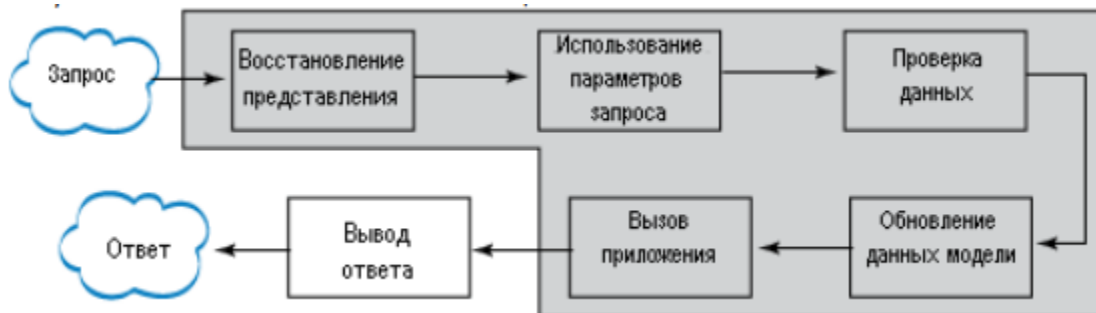
```
<c:forEach var="clip" items="${clipList}" >  
  <c:out value="${clip}">  
</c:forEach> (изменено)
```

## 7 Билет

### 1) AJAX в JSF

Идея заключается в том, что можно указать компоненты, которые JSF будет выполнять на сервере, а также компоненты, которые JSF будет перерисовывать при возвращении AJAX-ответа. Это делается с помощью появившегося в JSF 2 тега `<f:ajax />`

Все это работает за счет разделения жизненного цикла JSF на две части.



### 2) JPA архитектура.

JPA - является источником для хранения бизнес-сущностей как реляционных сущностей.

Единицы	Описание
EntityManagerFactory	Это фабричный класс EntityManager. Он создает и управляет несколькими экземплярами EntityManager.
EntityManager	Это интерфейс, он управляет операциями сохранения на объектах. Это работает как фабрика для экземпляра Query.
сущность	Сущности — это постоянные объекты, хранящиеся в виде записей в базе данных.
EntityTransaction	Он имеет непосредственное отношение к EntityManager. Для каждого EntityManager операции поддерживаются классом EntityTransaction.
Упорство	Этот класс содержит статические методы для получения экземпляра EntityManagerFactory.
запрос	Этот интерфейс реализуется каждым поставщиком JPA для получения реляционных объектов, соответствующих критериям.

(от себя добавь, ты ведь это в лабе писал...)

3) Компонент на React, у которого есть логин и пароль, и как лучше отправить. На стороне сервера - Rest API (Авторизация)

```
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(
      {
        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>
      }
    )
  }
}
```



## 8 Билет

### 1) Управляемые бины, конфигурация и чем XML лучше чем Аннотация

**Конфигурация контекста** задается через faces-config.xml или с помощью аннотаций 6 видов (внедрение аннотации выполняется до внедрения XML. Таким образом конфигурации XML будут иметь приоритет над аннотациями):

@NoneScoped - контекст не определен, жизненным циклом управляют другие бины.

@RequestScoped (по умолчанию) - контекст-запроса

@ViewScoped - контекст-страницы

@SessionScoped - контекст-сессии

@ApplicationScoped - контекст-приложения

@CustomScoped - бин сохраняется в Map, программист сам управляет его жизненным циклом

### 2) Как правильно описать класс с JPA

Entity — простой Java-класс (POJO), удовлетворяющий следующим требованиям

- 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.

**3) Написать на Angular проверка айдишника сессии у пользователя, что он какой-то определенный и если он не такой вывести что-то пользователю**

```
export class Component implements OnInit {
    @Input("token")
    token: string
    constructor (
        private messageService: MessageService,
        private jwtService: JWTService
    )
    // void | После того как Ангуляр инициализирует все свойства директивы, связанные с
    // данными вызовется метод для дополнительных задач инициализации.
    ngOnInit(): {
        if (!jwtService.isValid(token)) {
            messageService.showErrorMessage("Вы обесрались");
        }
    }
}
```

## 9 Билет

**1) Location Transparency, реализация в Java EE**

**Location Transparency** - реальное местоположение файла не важно.

Использование имен (например) для идентификации ресурсов вместо их фактического расположения.

Например: доступ к файлу предоставляется по уникальному имени, а сам файл может быть расположен где-то на жестком диске.

Основное преимущество является то, что мы не привязываемся к расположению ресурсов. Не важно где они находятся, обращение к ним одинаковое.

Контейнер будет таскать объекты с помощью RMI.

Для клиента создается видимость целостности приложения, как будто оно не расположено по разным серверам с разными JVM, то есть для использования Remote EJB ему не надо делать дополнительных движений.

## 2) Rest в Spring. Spring RESTful

**REST (RESTful)** - это общие принципы организации взаимодействия приложения/сайта с сервером посредством протокола HTTP. Особенность REST в том, что сервер не запоминает состояние пользователя между запросами - в каждом запросе передается информация, идентифицирующая пользователя (например, token, полученный через OAuth-авторизацию) и все параметры, необходимые для выполнения операции.

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

```
1 @ManagedBean(name="SampleBean")
2 public class SampleBean implements Serializable {
3
4     @ManagedProperty(value="#{AnotherSampleBean}")
5     private AnotherSampleBean anotherSampleBean;
6
7 }
```

## 10 Билет

### 1) Angular шаблоны и т.п.

Посмотри и расскажи своими словами на картинках все понятно и главные части:

```
export interface SocialMedia {  
  link:string,  
  user:string,  
  company?:boolean  
}
```

создаем html файл нашего компонента, и прописываем элементы как будет выглядеть наш вывод вывод соц сети

```
import { Component, Input } from '@angular/core';  
import { SocialMedia } from './social-media';  
  
@Component({  
  selector: 'ces-social-media',  
  templateUrl: './social-media.component.html',  
  styleUrls: ['./social-media.component.scss']  
})  
export class SocialMediaComponent {  
  @Input() facebook:SocialMedia;  
  @Input() twitter:SocialMedia;  
  @Input() linkedIn:SocialMedia;  
  @Input() medium:SocialMedia;  
  @Input() vertical:boolean = true;  
  
  facebookLink:string = "https://www.facebook.com/";  
  twitterLink:string = "https://twitter.com/";  
  linkedInLink:string = "https://www.linkedin.com/";  
  linkedInLinkCompany:string = "https://www.linkedin.com/company/"  
  mediumLink:string = "https://medium.com/@";
```

Вставляем например в footer нашей страницы

```
<ces-social-media ngClass="social-media-footer" [facebook]="{link:  
'egor_redgry', user: 'redgry' }" />
```

## 2) Dependency Lookup

Инверсия управления — один из популярных принципов объектно-ориентированного программирования, при помощи которого можно снизить связанность между компонентами, а также повысить модульность и расширяемость ПО.

В случае с поиском зависимостей класс должен самостоятельно реализовывать логику получения зависимостей извне. Для этого он должен иметь доступ к некоему источнику зависимостей.

поиск зависимостей предполагает, что компонент, для которого он применяется, должен реализовывать логику получения необходимых зависимостей.

Но в некоторых случаях использование поиска зависимостей всё же может быть более выгодно, например, при реализации шаблона проектирования «команда».

Команда — это поведенческий паттерн проектирования, который преобразует запросы в объекты, позволяя передавать их как аргументы при вызове методов, ставить запросы в очередь, логировать их, а также поддерживать отмену операций.

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

```
<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>*.xhtml</url-pattern>
  </servlet-mapping>
  <servlet-mapping>
    <servlet-name>Faces Servlet</servlet-name>
    <url-pattern>/faces/*</url-pattern>
  </servlet-mapping>
```

## 1) Валидаторы JSF

**Валидаторы в JSF** - нужны для того, чтобы подготовить данные для обновления объектов модели. Таким образом, к моменту вызова методов, реализующих логику приложения, можно сделать определенные выводы о состоянии модели.

**Стандартные валидаторы:**

- **DoubleRangeValidator**: Проверяет, что значение компонента укладывается в интервал, определяемый нижней границей, верхней границей или и тем, и другим. Значение должно быть числом.
- **LongRangeValidator**: Проверяет, что значение укладывается в интервал, определяемый нижней границей, верхней границей или и тем, и другим. Значение должно быть числом, преобразуемым к типу long.
- **LengthValidator**: Проверяет, что длина значения укладывается в интервал, определяемый нижней границей, верхней границей или и тем, и другим. Значение должно быть типа String.

**Для создания валидатора необходимо сделать следующее:**

- класс, реализующий интерфейс Validator (javax.faces.validator.Validator).
- Реализовать метод validate().
- Зарегистрировать валидатор в файле faces-config.xml или аннотацией @FacesValidator
- Использовать тег на страницах JSP.

## 2) JPA и запросы к бд. JPQL и Criteria API

JPA позволяет помещать объекты в БД и читать их.

JPQL используется для написания запросов к сущностям, хранящимся в реляционной БД. Он похож на SQL, но оперирует запросами, составленными по отношению к сущностям JPA, в отличие от прямых запросов к таблицам БД.

**JPA-Java-стандарт, который определяет:**

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

Запросы могут быть написаны на SQL или JPQL.

Для создания запроса:

- createQuery();
- createNamedQuery();
- createNativeQuery().

Для получения результата:

- `getSingleResult();`
- `getResultList()`

#### Criteria API (JPA 2.0)

Характерные черты:

- Объектно-ориентированный API для построения запросов.
- Есть возможность отобразить любой JPQL запрос в Criteria.
- Поддерживает построение запросов в runtime.

#### Создание Criteria Query

- Необходимо указать сущности, участвующие в запросе (query roots).
- Условие запроса задается через `where(Predicate p)`, где аргумент устанавливает необходимые ограничения.
- Метод `select()` определяет, что мы получим в результате запроса.

```
Root customer = qdef.from(Customer.class);
qdef.select(customer).where(queryBuilder
    .equal(customer.get("customerInfo"), ci));
```

- 3) EJB, который списывает деньги с клиента и отправляет их на счет банка за 1 транзакцию.

```
@Stateful
@TransactionManagement(BEAN)
public class MySessionBean implements MySession {
    @Resource UserTransaction ut;
    public void method() {
        try {
            ut.begin(); // Открываем транзакцию
            //... какие-то действия
            ut.commit(); // Закрываем транзакцию
        } catch (Exception e) {
            ut.rollback(); // Ошибка – откат транзакции
        }
    }
}
```

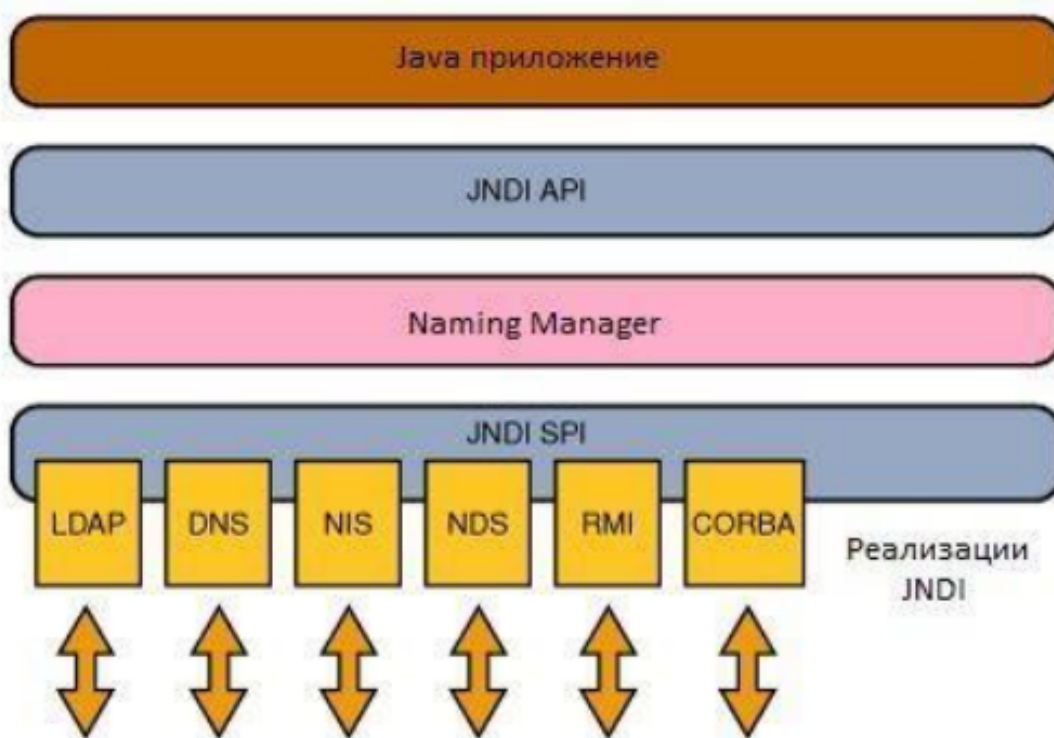


### 1) JNDI в Java EE

**Java Naming & Directory Interface** - это набор Java API, организованный в виде службы каталогов, который позволяет Java-клиентам открывать и просматривать данные и объекты по их именам, независимым от ресурсов способом.

Наиболее распространенный вариант использования - настройка пула соединений с базой данных на сервере приложений Java EE. Любое приложение, развернутое на этом сервере, может получить доступ к нужным соединениям, используя имя JNDI, `java:comp/env/FooBarPool` без необходимости знать подробности о соединении.

**Вы можете минимизировать** количество людей, которым необходимо знать учетные данные для доступа к производственной базе данных. Только сервер приложений Java EE должен знать, используете ли вы JNDI.



## 2) React, архитектура приложений на React

React - это JS-библиотека для разработки пользовательского интерфейса. Все в React это компоненты и обычно принимают форму JS классов. Вы можете создать свой компонент, создавая дочерний класс от React-Component класса.

### Ключевые особенности:

- декларативность
- универсальность (можно использовать на сервере и на мобильных платформах с помощью React Native)
- компонентный подход
- виртуальный DOM
- использование JSX (вставлять куски кода прямо в разметку HTML и связывать верстку и данные React)
- данные хранятся с помощью Redux, который представляет собой единое хранилище

## 3) Бин, который выводит количество минут, прошедших со старта сервера.

```
@ManagedBean
@ApplicationScoped
public class CounterBean implements Serializable {
    long startTime;

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

## 13 Билет

### 1) EJB

**EJB** - это технология разработки серверных компонентов, реализующих бизнес-логику. Жизненным циклом управляет EJB-контейнер (в составе сервера приложений).

**Аннотация @EJB** 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 другом JVM) объектам.

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.

#### **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 придется сериализовывать, что неплохо ускорит работу приложения.

Чтобы не делать бизнес-интерфейс, нужно пометить его аннотацией @LocalBean. А вот Remote бины имеют все прелести RMI: передачу по значению а не по ссылке, а значит нужна сериализация + обязательные бизнес-интерфейсы.

### 2) DI Angular

DI — важный паттерн дизайна приложения.

**dependency injection** это мощный механизм, при котором класс получает объект сервиса в конструкторе компонента и затем использовать по необходимости.

Чтобы задействовать сервис в компоненте, его нужно импортировать, добавить в коллекцию коллекцию providers. Тем самым определяется провайдер сервиса, который будет использоваться для его получения.

И после этого мы сможем задействовать встроенный в Angular механизм **dependency injection**.

### 3) Вывести данные из бина в JSF

Создаем класс с `@ManagedBean`, указываем его название, устанавливаем `@Scoped`, создаем внутри него поля, которые мы должны получать, устанавливаем сетеры и гетеры, создаем пустой конструктор  
Ну и уже в файле `.xhtml` где нужно нам выводим наши поля, с помощью EL-выражений используя доллар для чтения `${mybean.firstName}`

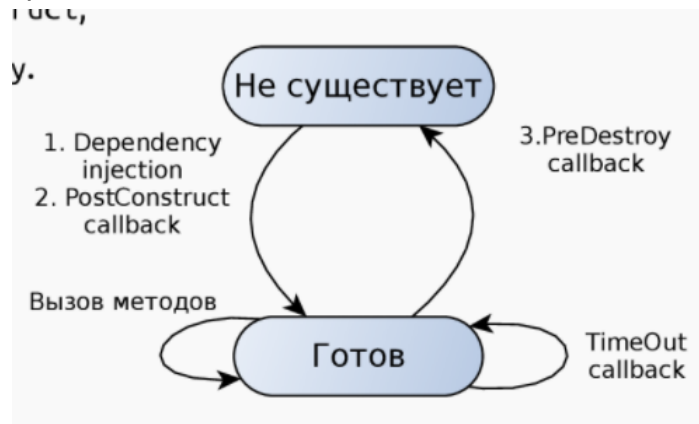
### 1) Session Beans, виды и жизненный цикл

**Sessions Beans** - предназначены для синхронной обработки вызовов. Вызываются посредством обращения через API. Могут вызываться локально и удаленно. Не обладают свойством персистентности. Можно формировать пулы бинов (за исключением @Singleton)

#### Виды:

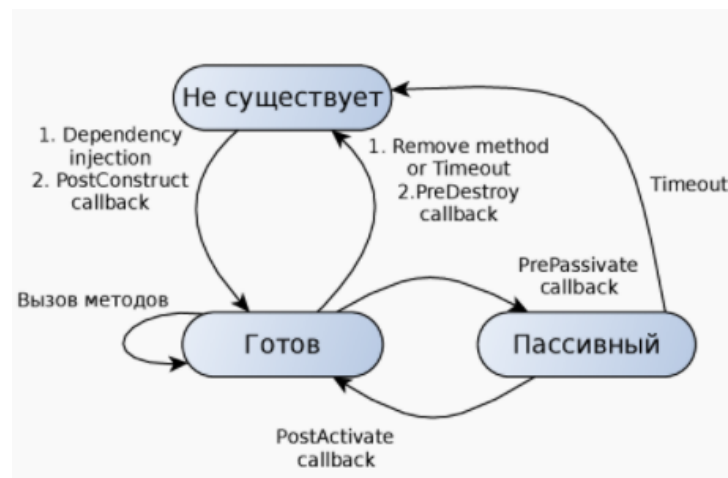
**@Stateless** - не сохраняет состояние между последовательными обращениями клиента. Нет привязки к конкретному клиенту. Хорошо масштабируются.

Цикл:



**@Stateful** - "Привязываются" к конкретному клиенту. Можно сохранять контекст в полях класса. Масштабируются хуже чем Stateless.

Цикл:



**@Singleton** - контейнер гарантирует существование строго одного экземпляра такого бина.

## 2) JSX в React

JSX - это расширение синтаксиса JavaScript, который позволяет использовать HTML-подобный синтаксис для описания структуры интерфейса. Как правило, компоненты написаны с использованием JSX, но также есть возможность использовать обычный JavaScript. (Смесь JS + HTML), Все потом компилируется в обычный JS при помощи инструмента Babel.

## 3) Навигация в JSF, сконфигурировать через XML

```
<h:commandButton action="otherPage" value="Submit" />
```

```
<navigation-rule>
```

```
  <from-view-id>/start.xhtml</from-view-id>
```

```
  <navigation-case>
```

```
    <from-outcome>otherPage</from-outcome>
```

```
    <to-view-id>/page.xhtml</to-view-id>
```

```
    <redirect/>
```

```
  </navigation-case>
```

```
</navigation-rule>
```

**1) EJB. Что такое Local и Remote**

**EJB** - это технология разработки серверных компонентов, реализующих бизнес-логику. Жизненным циклом управляет EJB-контейнер (в составе сервера приложений).

**Аннотация @EJB** 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 другом JVM) объектам.

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.

**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 придется сериализовывать, что неплохо ускорит работу приложения.

Чтобы не делать бизнес интерфейс, нужно 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.

## 2) SPA (Single page app) + плюсы и минусы

Все веб приложения делятся на одностраничные (SPA) и много страничные (MPA). SPA - это веб-приложение, в которых загрузка необходимого кода происходит на одну страницу. Это позволяет сэкономить время на повторную загрузку одних и тех же элементов. Все элементы загружаются при инициализации. Также данный вид приложения загружает дополнительные модули после запроса от пользователя.

### Плюсы:

- 1) Приложения на SPA отлично работают на стационарных устройствах, так и на мобильных
- 2) Скорость работы выше

### Минусы:

- 1) Сильно усложняет Frond-End, Кроме HTML и логики UI тут ещё и роутеры, MVC и прочие сладости.
- 2) Так как у приложения одна точка входа, существует риск того, что одна ошибка может привести к нерабочему состоянию всего приложения.
- 3) Дублирование роуторов

## 3) Criteria Api. Выбрать всех студентов, у которых балл ниже 4 и удалить

```
@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();
    }
}
```



## 16 Билет

### 1) Компоненты Angular

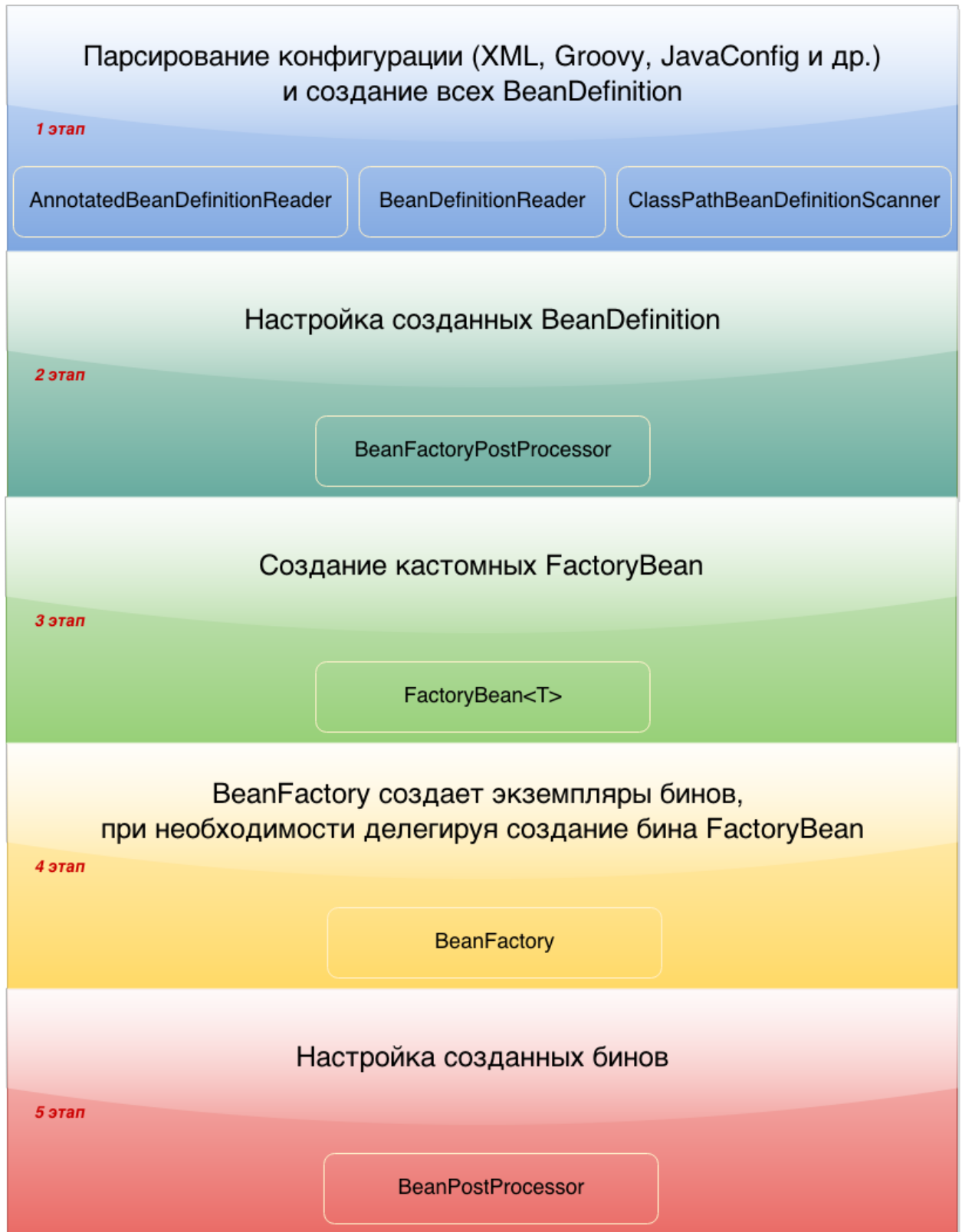
С помощью компонентов в Angular происходит управление отображением представления на экране. Для того, чтобы класс мог использоваться в других модулях, он определяется с ключевым словом *export*. Для создания компонента нужно с помощью аннотации `@Component` (это функция декоратора из библиотеки `@angular/core`) Декоратор позволяет идентифицировать класс как компонент. Мы в нем можем указать свойство `selector`, который можем в будущем использовать в html файле. Указать URL на шаблон нашего компонента (.html файл) и стили если имеются (Это представление компонента).

## 2) Spring инициализация

Все бины типа *MyBean*, получаемые из контекста, будут создаваться с уже проинициализированными полями `value1` и `value2`. Также тут стоит отметить, этап на котором будет происходить инжект значений в эти поля будет зависеть от того какой `@ Scope` у вашего бина.

SCOPE\_SINGLETON — инициализация произойдет один раз на этапе поднятия контекста.

SCOPE\_PROTOTYPE — инициализация будет выполняться каждый раз по запросу. Причем во втором случае ваш бин будет проходить через все `BeanPostProcessor`-ы что может значительно ударить по производительности.



**СПУСТИСЬ НА 2 ВОПРОС 21 БИЛЕТА!!!**

### 3) React. Поля ввода для банковской карты.

```
import {React} from 'react'
export class CreditCard extends React.Component{

  constructor(props){
    super(props)
    this.state = {
      number = "",
      data = "",
      owner = "",
      CVV = ""
    }
    //Так же все методы надо привязать,чтобы использовать this в них, приведу один
    this.onChange = this.onChange.bind(this)
  }

  //Все аналогичны, напишу один
  onChange(e) {
    //Вот тут делаете валидацию или добавляете / если это Data
    this.state.number = e.value
  }

  onClick(e) {
    sendData(this.state.number, this.state.data, this.state.CVV, this.state.owner)
  }

  render() {
    return(
      <input onChange={this.onChangeNumber(event)} value={this.state.number}/>
      <input onChange={this.onChangeData(event)} value={this.state.data}/>
      <input onChange={this.onChangeOwner(event)} value={this.state.owner}/>
      <input onChange={this.onChangeCVV(event)} value={this.state.CVV}/>
      <button onClick={this.onClick()} />
    )
  }
}
```

## 17 Билет

### 1) ORM проблемы и решения

ORM — Object/Relational Mapping — преобразование данных из объектной формы в реляционную и наоборот. Сложности, возникающие при попытке отобразить один вид представления данных на другой, называют объектно-реляционным несоответствием

Основные проявления объектно-реляционного несоответствия:

1) Проблема идентичности.

2) Представление наследования и полиморфизма.

Три способа реализации:

Одна таблица для всех классов.

Плюсы: простота и производительность

Минусы: отсутствие null ограничений, не нормализованная таблица

Своя таблица на каждый класс

Плюсы: возможность null ограничений

Минусы: плохая поддержка полиморфных записей, не нормализованная таблица

Своя таблица на каждый подкласс

Плюсы: нормализованная таблица, возможность null ограничений

Минусы: низкая производительность

3) Проблема навигации между данными.

## 2) Angular модули компоненты сервисы DI

Angular предоставляет такую функциональность, как двустороннее связывание, позволяющее динамически изменять данные в одном месте интерфейса при изменении данных модели в другом, шаблоны, маршрутизация и так далее.

Одной из ключевых особенностей Angular является то, что он использует в качестве языка программирования TypeScript Angular CLI – инфраструктура, упрощает создание приложения, его компиляцию.

Приложение Angular состоит из модулей. Главная цель модуля - группирование компонентов и/или сервисов, связанных друг с другом. Это позволяет легко подгружать и задействовать только те модули, которые непосредственно необходимы. И каждое приложение имеет как минимум 1 корневой модуль.

Сервисы в Angular представляют широкий спектр классов, которые выполняют некоторые специфические задачи, например логгирование, работу с данными и т.д.

В отличие от компонентов и директив сервисы не работают с представлениями, то есть с разметкой html, не оказывают на нее прямого влияния. Они выполняют определенную и достаточно узкую задачу.

**DI в Angular**

DI — важный паттерн дизайна приложения.

**dependency injection** это мощный механизм, при котором класс получает объект сервиса в конструкторе компонента и затем использовать по необходимости.

Чтобы задействовать сервис в компоненте, его нужно импортировать, добавить в коллекцию коллекцию providers. Тем самым определяется провайдер сервиса, который будет использоваться для его получения.

И после этого мы сможем задействовать встроенный в Angular механизм **dependency injection**.

## 3) Spring data табличку из студентов какой-то хуйни добавление удаление обновление короче нихуя не понял

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

### 1) Что такое транзакции в Java EE. JTA

Транзакции это разделение выполнения процессов приложения сразу с несколькими JVM. Их можно открывать и закрывать.

JTA предоставляет высокоуровневый интерфейс для управления транзакциями (begin, commit, rollback), избавляя от необходимости работы с каждым ресурсом по-своему.

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`

### 2) Навигация в React

Router определяет набор маршрутов и, когда к приложению, приходит запрос, то Router выполняет сопоставление запроса с маршрутами. И если какой-то маршрут совпадает с URL запроса, то этот маршрут выбирается для обработки запроса.

Для выбора маршрута определен объект `Switch`. Он позволяет выбрать первый попавшийся маршрут и его использовать для обработки. Без этого объекта Router может использовать для обработки одного запроса теоретически несколько маршрутов, если они соответствуют строке запроса.

Каждый маршрут представляет объект `Route`. Он имеет ряд атрибутов. В частности, для маршрута устанавливаются два атрибута:

`path`: шаблон адреса, с которым будет сопоставляться запрошенный адрес URL

`component` - тот компонент, который отвечает за обработку запроса по этому маршруту

`Switch` - для поиска первого совпавшего пути (прим. сначала идет `"/"`, потом `"/main"`, мы ищем `"/main"`, открывает `"/"`)

`BrowserHistory` - сохраняется в историю, имеет человеческие названия

`HashHistory` - не сохраняется, имеет хэши в URL

```
<Router> <Switch>
  <Route exact path="/" component={Main} /> <Route path="/about" component={About} />
  <Route component={NotFound} />
</Switch> </Router>
```

### 3) EJB, который в 12 ночи выведет содержимое таблицы

Выводит в полночь столбец name из таблицы Persons

@Singleton

```
public class Test implements TestInterface{
    @PersistenceContext(unitName = "mypersistence-unit")
    private EntityManager em;

    @Schedule(hour="0", minute="0", second="0")
    public void midnightMethod(){
        em.createQuery("SELECT * FROM PERSONS")
            .getResultList()
            .forEach(p->System.out.println(p.name));
    }
}
```



**1) Facelets**

Facelets - это фреймворк, который требует для своей работы валидные XML документы. Он включает в себя теги, которые можно использовать например для создания шаблона страницы и встраивания в него компонентов, которые мы прописали в отдельных файлах или другие вещи. Facelets - поддерживает все компоненты JSF и создает собственное дерево компонентов.

Наименование тега	Описание
ui:include	Включает содержимое из другого файла XML
ui:composition	Будучи использованным без атрибута template, этот тег определяет последовательность элементов, которая может быть вставлена в другом месте. Композиция может иметь переменные части указанные с помощью дочерних тегов ui:insert. При использовании с атрибутом template, загружается шаблон. Дочерние теги этого тега определяют переменные части шаблона. Содержимое шаблона заменяет этот тег.
ui:decorate	Будучи использованным без атрибута template этот тег определяет страницу, в которую могут быть вставлены части. Переменные части задаются с помощью дочерних тегов ui:insert. При использовании с атрибутом template, загружается шаблон. Дочерние теги этого тега определяют переменные части шаблона.
ui:define	Определяет содержимое, которое вставляется в шаблон с помощью соответствующих тегов ui:insert.
ui:insert	Вставляет содержимое в шаблон. Это содержимое определяется в теге, который загружает шаблон.
ui:param	Задаёт параметр, передаваемый во включенный файл или шаблон.
ui:component	Этот тег идентичен ui:composition, за исключением того, что создает компонент добавляемый к дереву компонентов.
ui:fragment	Этот тег идентичен ui:decorate, за исключением того, что создает компонент добавляемый к дереву компонентов.
ui:debug	Этот тег позволяет пользователю с помощью определенной комбинации клавиш вывести на экран окно отладки, в котором показаны иерархия компонентов для текущей страницы и переменные с областью действия приложения.
ui:remove	Реализация JSF удаляет все, что находится в этом теге
ui:repeat	Выполняет итерации по списку, массиву, результирующему набору или отдельному объекту.

## 2) JPA vs JDBC

Основное отличие между ними является уровень абстракций. JDBC - это стандарт низкого уровня для взаимодействия с БД. JPA - это стандарт более высокого уровня с той же целью. JPA позволяет использовать объектную модель в своем приложении, которая может сделать вашу жизнь намного проще. JDBC - позволяет вам делать больше вещей с БД напрямую, но это требует большего внимания. Некоторые задачи не могут быть эффективно решены с использованием JPA, но могут быть решены более эффективно с JDBC (т.к. Ты сам пропишешь как должно работать).

## 3) Форма ПСЖ на Angular

Пример (не псж, но как нужно делать):

Создаем файл html, в котором прописываем шаблон нашей формы с нужными нам полями. После этого создаем на TypeScript файл, в котором прописываем наш компонент, указываем его селектор, наш шаблон формы ПСЖ, ну и гейские стили обязательно. В самом классе компонента, создаем форму с помощью `formBuilder.group(...)`, в котором указываем поля, которые будет заполнять пользователь.

Создаем конструктор с `formBuilder: FormBuilder`. Создаем метод `onSubmit()` и прописываем в нем логику, что мы будем делать с нашей заполненной формой, выводить в консоль или куда-то отправлять. Ну и вывод сообщения и очистку

форму не забудем добавить. Сбросив форму через метод reset()

```
import { Component } from '@angular/core';
import { FormBuilder } from '@angular/forms';

import { CartService } from '../cart.service';

@Component({
  selector: 'app-cart',
  templateUrl: './cart.component.html',
  styleUrls: ['./cart.component.css']
})
export class CartComponent {
  items = this.cartService.getItems();
  checkoutForm = this.formBuilder.group({
    name: '',
    address: ''
  });
  constructor(
    private cartService: CartService,
    private formBuilder: FormBuilder,
  ) {}

  onSubmit(): void {
    // Process checkout data here
    this.items = this.cartService.clearCart();
    console.warn('Your order has been submitted', this.checkoutForm.value);
    this.checkoutForm.reset();
  }
}
```

20 Билет

## 1) Sessions Beans

**Sessions Beans** - предназначены для синхронной обработки вызовов. Вызываются посредством обращения через API. Могут вызываться локально и удаленно. Не обладают свойством персистентности. Можно формировать пулы бинов (за исключением @Singleton)

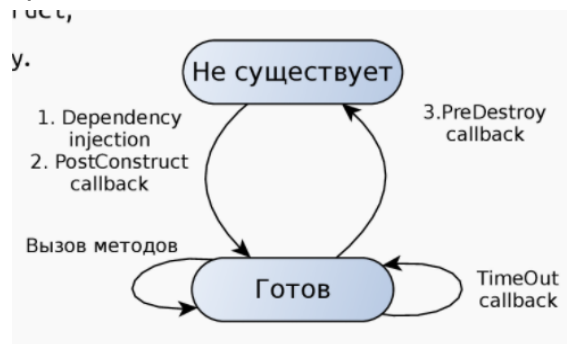
### Требования к классам Session Beans

- Должен быть public классом верхнего уровня.
- Не должен быть final и/или abstract.
- Должен иметь public конструктор без параметров.
- Не должен переопределять метод finalize().
- Должен реализовывать все методы бизнес-интерфейса(-ов).

### Виды:

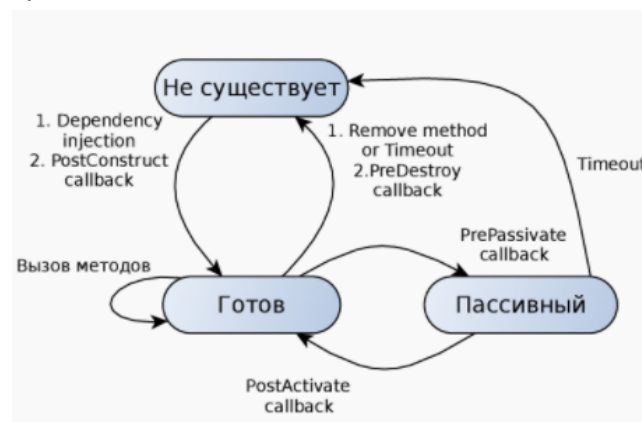
@Stateless - не сохраняет состояние между последовательными обращениями клиента. Нет привязки к конкретному клиенту. Хорошо масштабируются.

Цикл:



@Stateful - "Привязываются" к конкретному клиенту. Можно сохранять контекст в полях класса. Масштабируются хуже чем Stateless.

Цикл:



@Singleton - контейнер гарантирует существование строго одного экземпляра такого бина.

## 2) React. Props & State. Глупые и умные компоненты

**State (состояние)** — набор данных, которые отражают состояние компонента в конкретный момент времени.

**Props (свойства)** — данные, передаваемые компоненту через атрибуты.

**props и state** обычные объекты JS.

**props и state** иницииируют выполнение функции render.

### Глупые компоненты:

1. не зависят от остальной части приложения, например Flux actions или stores часто
2. содержатся в this.props.children
3. получают данные и колбэки исключительно через props
4. имеют свой css файл
5. изредка имеют свой state
6. могут использовать другие глупые компоненты примеры: Page, Sidebar, Story, UserInfo, List

### Умные компоненты:

1. оборачивает один или несколько глупых или умных компонентов
2. хранит состояние стора и пробрасывает его как объекты в глупые компоненты
3. делают запросы на сервер

Действия простых компонентов React просты и однообразны, они всего лишь выводят данные, принимаемые ими от свойств. Умные компоненты управляют простыми, делают запросы на сервер и многое другое.

## 3) JSF. Валидация многострочного инпута только латинских символов.

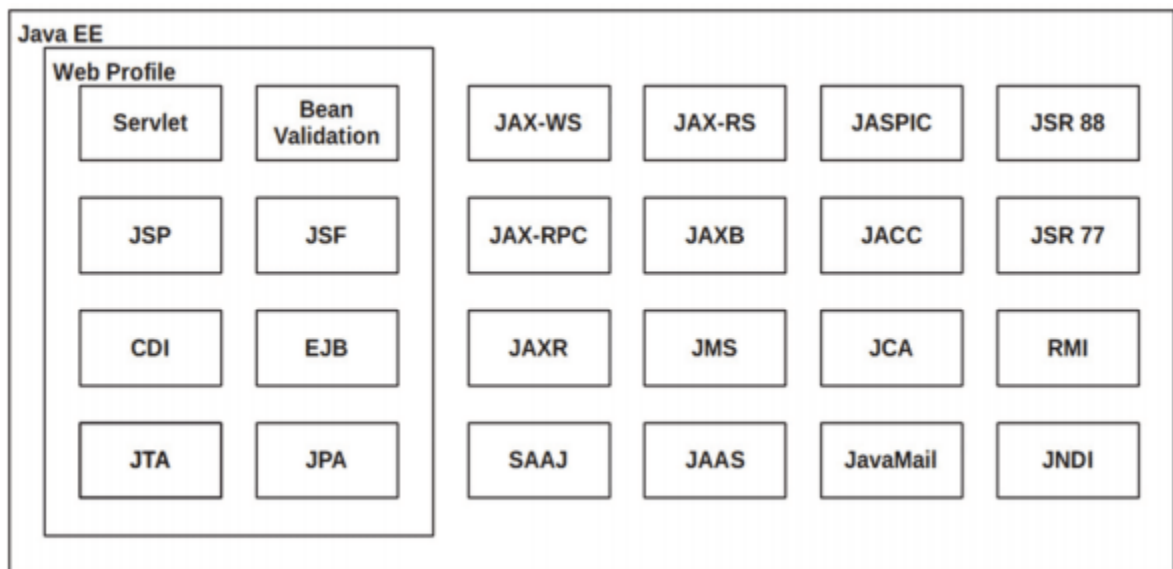
```
<h:inputTextArea><f:validateRegex pattern="[a-z]*/></h:inputTextArea>
```

### 1) Платформа Java EE

Java Enterprise Edition - представляет платформу для создания корпоративных приложений на языке Java. Прежде всего это сфера веб-приложений и веб-сервисов.

Java EE - состоит из набора API и среды выполнения. Примерами API являются: Java Servlets, JSP, EJB и другие.

- Появились в Java EE 6.
- Позволяют 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.



## 2) Трехфазные и двухфазные конструкторы в Java EE и Spring

Двухфазный конструктор заключается в том, что мы в конструкторе вызываем фазу 1 (за который отвечает java), а в метод init() фазу 2 указав аннотацию @postConstruct (за который отвечает BeanPostProcessor). В конструкторе у нас не будет проинициализирована переменная, а уже в методе init() будет. Двойной проход по бин процессорам

Трехфазный конструктор заключается в аннотации @PostProxy (за который отвечает Context Listener) она выполнится самой последней

Главное не забыть зарегистрировать аннотации в файле context.xml

## 3) JSF страница, которая показывает первые 19 простых чисел и с помощью ајах подгружает другие 10

(Пример, переделать и сказать что на стороне сервера у нас при нажатии кнопки к списку из бд например подгружаются 10 чисел и обновляется с помощью рендера)

```
<h:commandButton id="submit-button"
    disabled="false"
    type="submit"
    action="#{dotsBean.addPoint}"
    value="Отправить на проверку"
    onclick="drawDotsFromTable()">
    <f:ajax execute="newEntryForm" render="valuesTable"/>
</h:commandButton>
<h:panelGroup id="valuesTable">
    <h:dataTable styleClass="table-check" value="#{dotsBean.dotsList}" var="dot">
```