## **Билет 1 +**

### **1. Фазы jsf: Invoke Application и Render Response**

При поступлении запроса необходимо выполнить определенную цепочку действий, чтобы проанализировать запрос и подготовить ответ. За программиста это делает фреймворк(JSF)

Жизненный цикл запроса ответа:

Восстановление представления

Получение параметров запроса

Проверка значений

Обновление свойств бинов

Обработка приложения

Формирования ответа

1) Invoke Application Phase

● Управление передаётся слушателям событий. JSF обрабатывает события, также решается вопрос навигации.

● Формируются новые значения компонентов.

(JSF обрабатывает события, которые были сгенерированы нажатием на кнопки и ссылки. На данном этапе также решаются вопросы, связанные с навигацией приложения, если это необходимо. Если один из компонентов формы имеет свойство immediate="true", то он должен был быть обработан в фазе "Применение значений запроса".)

Вызывается метод UIViewRoot#processApplication() для обработки событий, у которых определен идентификатор фазы PhaseId.INVOKE\_APPLICATION.

Перехватить обработку событий можно переопределением ActionListener.

2) Render Response Phase

● JSF Runtime обновляет представление в соответствии с результатами обработки запроса.

● Если это первый запрос к странице, то компоненты помещаются в иерархию представления.

● Формируется ответ сервера на запрос.

● На стороне клиента происходит обновление страницы.

(JSF создает ответ, основываясь на данных, полученных на предыдущих шагах. Информация страницы обновляется данными из managed bean и генерируется html страница с помощью Renderers. Если на предыдущих шагах происходили какие-либо ошибки, то они инкапсулируются в тег .)

Render Response - это завершающая фаза жизненного цикла JSF, в которой формируется ответ. Ответ формируется каскадно, как и во всех предыдущих фазах : вызываются методы encodeXX() каждого компонента. Эти методы определяют, как компоненты (точнее, их визуализаторы — renderers) будут отображаться клиенту. Язык разметки ответа может быть любым : HTML, XHTML, XML и т.п. Кроме генерации ответа на данном этапе происходит сохранение текущего состояния вида, обеспечивая возможность его восстановления при последующих обращениях к странице.

Таким образом, в фазе Render Response фреймворк выполняет два действия:

* формирует ответ для клиента;
* сохраняет состояние результата для последующий запросов.

По завершении рендеринга конечное состояние вида должно быть сохранено посредством методов класса StateManager. Эта информация о состоянии должна быть доступна при последующем запросе, чтобы ей можно было воспользоваться в фазе Restore View.

К фазе Render Response также относится и объединение статического описания страницы («шаблон»), с ее динамическим наполнением. Но это уже связано и с используемыми фреймворками типа Facelets, Tiles.

### **2. Способы задания конфигурации в Spring**

1. XML-файл, в котором вручную прописываются все бины, путь до класса, свойства, конструкторы. Загружается этот файл из classpath (можно запихать в ресурсы jarника). способ конфигурации спринг это конфигурация с помощью конфиг файла applicationContext.xml.  
   В него в ручную прописываются все бины, создание объектов , их типы и т.д. Далее этот файл нужно указать при создании ClassPathXmlApplicationContext.
2. (class based config). Создаем класс с аннотацией Configuration, внутри помечаем методы аннотацией Bean.
3. (annotation based config) добавляя к классам аннотации Component и т.д. Внедрение зависимостей происходит через аннотацию Autowired.
4. Groovy config
5. Property files (старый варик)

2 и 3- java config

### **3. Написать исходный код CDI бина, реализующего паттерн «команда» (Command Pattern)**

interface Command {

void execute();

}

@Named(value=”cmd1”)

@ApplicationScoped

public class Cmd1 implements Command {

void execute() { ... };

}

@Named(value=”cmd2”)

@ApplicationScoped

public class Cmd2 implements Command {

void execute() { ... };

}

@Named

@ApplicationScoped

public class MyBean implements Command {

private final Command cmd1, cmd2;

@Inject

public MyBean(@Named(“cmd1”) Command cmd1, @Named(“cmd2”) Command cmd2) {

this.cmd1 = cmd1;

this.cmd2 = cmd2;

}

public void cmd(int n) {

if (n == 1) cmd1.execute();

if (n == 2) cmd2.execute();

}

}

## **Билет 2**

### **1. Принципы IoC и CDI. Реализация в Java EE**

Инверсия управления — принцип, используемый для уменьшения связности кода. Заключается в том, что самописным кодом управляет общий фреймворк, занимающийся жизненным циклом компонентов и коммуникацией между ними. Чаще всего реализуется посредством внедрения зависимостей.

Inversion of Control (инверсия управления) — принцип работы программы, который диктует нам как писать слабо связанный код. Он говорит, что компонент системы должен быть как можно более изолированным от других. Так же есть IoC контейнер, который упрощает работу с компонентами, автоматизирует написание кода, в общем по максимуму берет работу с компонентами на себя.

● Жизненным циклом компонента управляет контейнер (а не программист).

● За взаимодействие между компонентами отвечает тоже контейнер.

Contexts and Dependency Injection(внедрение зависимостей) — позволяет снизить (или совсем убрать) зависимость компонента от контейнера (контейнер не порождает компонент: логика взаимодействия с контейнером описывается аннотациями, программист не зависит от API контейнера).

● Не требуется реализации каких-либо интерфейсов.

● Не нужны прямые вызовы API.

● Реализуется через аннотации. В жава ее для впрыскивания @Inject, для выдачи имени @Named

IoC и CDI реализованы в JavaEE.

Сперва, нужно создать файл beans.xml, куда прописать настройку всех созданных бинов. А далее с помощью аннотации Inject внедрить зависимость в конструкторы.

Также у бинов есть скопы (Request, Session, Application и тд), по которым контейнер, понимает когда уничтожать или создавать новый бин.

### **3. JSF страница, динамически подгружаемая и выводящая новостную ленту с новостями формата: автор, заголовок, дата, иллюстрация, аннотация и полный текст(показывается при нажатии на соответствующую строчку)**

<!DOCTYPE html>

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

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

xmlns:f="http://xmlns.jcp.org/jsf/core">

<h:head>

<title>Новостная лента</title>

</h:head>

<h:body>

<h:dataTable var="item" value="#{newsFeed.news}">

<h:column>

<h:outputText value="#{item.author}"/>

<br/>

<h:outputText value="#{item.header}"/>

<br/>

<h:outputText value="#{item.date}"/>

<br/>

<h:graphicImage url="#{item.img}"/>

<br/>

<h:outputText value="#{item.annotation}"/>

<br/>

<h:form>

<h:commandButton action="#{item.toggleShow()}" value="Показать текст"/>

<br/>

<h:outputText value="#{item.fullText}" rendered="#{item.show}"/>

</h:form>

</h:column>

</h:dataTable>

</h:body>

</html>

## **Билет 3 +**

### **1. Реализация шаблона MVC в JSF**

1) Model — бины, в которых содержится бизнес логика. Model в JSF это managed bean, джава классы, они содержат бизнес-логику и передаются юзеру. К ним указываются аннотации, с помощью которых настраивается их поведение.

● Содержат параметры и методы для обработки данных с компонентов.

● Используются для обработки событий UI и валидации данных.

● Жизненным циклом управляет JSF Runtime Envronment.

● Доступ из JSF-страниц осуществляется с помощью элементов EL.

● Конфигурация задаётся в faces-confg.xml (JSF 1.X), либо с помощью аннотаций (JSF 2.0).  
2) View — xhtml шаблон в котором формируется дерево компонентов. Компоненты могут взаимодействовать с бинами, вызывать их методы или получать из них данные, тем самым передавая их пользователю. View в JSF это верстка, написанная в расширении xhtml. Это очень удобный формат,так как он понятен браузеру, а также его можно расширять, добавляя библиотеки.  
3) Controller — реализуется самим фреймворком. Это класс FacesServlet, который занимается диспетчеризацией и управлением жизненным циклом. Controller в JSF мы не пишем руками, он реализуется фреймворком, нам нужно лишь указывать в верстке action’ы по которым фреймворк будет понимать что мы хотим получить.

● Обрабатывает запросы с браузера.

● Формирует объекты-события и вызывает методы-слушатели.

### **2. Основные аннотации Java EE CDI**

У CDI Bean’ов есть множество аннотаций:

- @RequestScoped - контекст - запрос;

- @ViewScoped - контекст-страница(компонент создается один раз при обращении к странице, и используется ровно столько, сколько пользователь находится на странице); из JSF, но тоже работает

- @SessionScoped - контекст - сессия;

- @ApplicationScoped - контекст - приложение;

- @ConversationScoped - Областью жизни управляет программист. Управление осуществляется через инъекцию объекта javax.enterprise.context.Conversation.

- @Dependent. Используется по умолчанию. ЖЦ определяется тем где он был использован

@Produces/@Disposes, бины- фабрики которые управляют экземплярами других бинов

@Informal — “бин-наследник”

@Inject - используется для указания точки внедрения зависимости.. Инъекции могут происходить в поле, в метод и в конструкторе

@Named - используется, для того, чтобы выдать имя бину, тогда его можно будет использовать на jsf странице. Как правило эта аннотация используется в совокупности с аннотацией скоупа

@Qualifier - аннотация, которая используется для создания аннотаций-спецификаторов, которые четко указывают, какой бин надо инжектить. Над классом ставится аннотация для указания квалификатора бина. Над точкой внедрения ставится такая же аннотация.

@Default/Alternative — управляет выбором бина при наличии нескольких

### **3. Написать React компонент формирующий таблицу пользователей, данные приходят в props**

function UsersTable(props) {

return (

<table className=”table”>

<thead>

<tr><td>Name</td><td>Surname</td></tr>

</thead>

<tbody>

{ props.data.map((user, i) => (

<tr key={i}>

<td>{user.name}</td>

<td>{user.surname}</td>

</tr>

))}

</tbody>

</table>

)

}

## **Билет 4 +**

### **Spring Web MVC: View Resolvers**

Реализация MVC в SPRING:

Диспетчер сервлет получает запрос, далее он смотрит в свои настройки, чтобы понять какому контроллеру передавать запрос.

Далее в контроллере происходит обработка запроса. После обработки ответ приходит на диспетчера.

На основании полученных данных, диспетчер ищет нужное ему представление (ViewResolver).

В представление передаются данные модели(или в модель, если это нужно), и представление посылается пользователю.

View Resolver — интерфейс, реализуемый объектами, которые способны находить представление по его имени. Так же возможна поддержка локализации.

● Представление в Spring Web MVC может быть построено на разных технологиях.

● С каждым представлением сопоставляется его символическое имя.

● Преобразованием символических имён в ссылки на конкретные представления занимается специальный класс, реализующий интерфейс org.springframework.web.servlet.ViewResolver.

● Существует много реализаций ViewResolver для разных технологий построения представления.

● В одном приложении можно использовать несколько ViewResolver'ов.

### **2. Angular: ключевые особенности, отличия от AngularJS**

● OpenSource-фреймворк.

● Первая версия выпущена в 2009 г.

● Предназначен для разработки SPA.

● Интерфейс строится из компонентов.

● Компоненты могут рекурсивно вкладываться друг в друга.

● Компоненты могут объединяться в библиотеки.

Angular - написана на TypeScript, развитие AngularJS

особенности:

-кроссплатформенное

-для разработки надо настроить сборочное окружение (node.js или npm)

-приложение состоит из модулей (NgModules)

-модули обеспечивают контекст для компонентов (components)

-из компонентов строятся представления (views)

-компоненты взаимодействуют с сервисами (services) через DI

Angular 2 как и AngularJS реализуют модель MVVM

Однако AngularJS в настоящее время является устаревшим, в нем были проблемы с производительностью, неупорядоченная документация и много лишних инструментов.

В AngularJS жесткие рамки для компонентов, их сложно использовать повторно, в Angular’е есть иерархия компонентов, что дает возможность легко и просто реюзать их.

AngularJS универсален, но гораздо менее безопасен и управляем.

AngularJS задействует JavaScript, Angular же использует TypeScript ( выше безопасность и в общем язык современнее).

Еще Angular адаптирован под слабые мобильные устройства(чего нет в AngularJS).

### **3. Написать веб-приложение на JSF (xhtml + CDI-бин) со списком студентов и бин, который будет реализовывать логику отчисления студентов. Напротив каждого имени студента должна быть кнопка "отчислить". Обновление должно производиться при помощи AJAX**

Java:

@Named(value = “student”)

@SessionScoped

public class Student implements Serializable {

private DBHelper dbhelper;

public void delete(username) {

dbhelper.removeByUsername(username);

}

public List<User> getList() {

return dbhelper.getStudents();

}

}

XHTML:

<!DOCTYPE html>

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

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

xmlns:f="http://xmlns.jcp.org/jsf/core"

xmlns:ui="http://xmlns.jcp.org/jsf/facelets">

<f:view>

<h:head>

...

<title>Студенто-отчилялка</title

...

</h:head>

<h:body>

<ui:repeat id="table" className="table" value="student.list" var="student” />

<div class=”row”>

<h:outputText value=”#{student.user.name}” />

<h:commandButton id=”#{student.user.name”} action=”#{student.delete(student.user.name)}”>

<f:ajax execute=”#{student.user.name”}” render=”table” />

</h:commandButton>

</div>

</ui:repeat>

</h:body>

</f:view>

</html>

## 

## **Билет 5 +**

### **1. Профили и платформы Java EE**

В JavaEE существует два профиля (появились в JAVA EE 6):

Web Profile — содержит в себе только те компоненты, которые нужны для работы веб приложения, это Servlet’ы, JSP, JSF, JPA, CDI, EJB.

Full Profile — полный сборник джавы ее, в нем по мимо Web есть еще JAX-RS, JAX-WS, JAXB, JNDI, JAVA MAIL и еще очень много всего.

Платформы:

Существует 3 самых известных API: JME, JSE, JEE

JME-(Java Micro Edition) представляет из себя API и минимально требовательную VM для разработки и старта приложения на смартфоне/планшете. Основана на более ранней версии JSE, поэтому некоторые функции (напр. дженерики) не работают

JSE-(Java Standard Edition) занимается обеспечением основными стандартными функциями самой Java, и она определяет абсолютно все: базовые типы и объекты языка, классы более высокого уровня, производительность приложения в сети и обеспечение защищенности. Включает в себя пакеты: java.util,java.math,java.io,java.nio,java.lang и тд.

JEE-(Java Enterprise Edition) для разработки Enterprise приложений. Она строится на основе платформы JSE, а еще дает возможность разработки более крупно масштабируемых , сложно уровневых и безопасных программ. Содержит: WebSocket, JSF, Unified EL, API для веб-служб RESTful, DI, EJB, JPA, и Java Transaction API.

### **2. Типы DI в Spring**

В Spring реализовано три типа внедрения зависимостей:

1) Constructor injection - контейнер вызовет конструктор с аргументами бинов, которые потом заинджектятся в класс. Через конструктор, для этого внедряемый класс должен быть помечен как компонент и его внедрение должно быть либо показано аннотацией autowired, либо прописано в конфигах

// bean code

@Autowired

public Car(Wheel wheel) {

setFirstVal(null);

}

// java config code

@Bean

public Car getCar() {

return new Car(firstWheel());

}

// xml config code

<bean id = "car" class = "ru.ifmo.cse.Car">

<constructor-arg name = "wheel" ref = "firstWheelBean" />

</bean>

2) Property injection - контейнер через рефлексию будет в поля класса пропихивать зависимости (не рекомендуется к использованию, рефлексия ест много ресурсов, очень легко нарушить Single Responsibility принцип, тяжело отлаживать)

// bean code

@Autowired private Wheel firstWheel;

// java config code

нет, т.к. внедрение происходит автоматически, используя рефлексию

// xml config code

<bean id = "car" class = "ru.ifmo.cse.Car" autowired = ”firstWheel” />

3) Setter injection.- сначала контейнер вызовет конструктор бина без аргументов, после вызовет помеченные аннотациями @Autowired сеттеры и впихнет туда нужные зависимости. Через сеттеры для этого внедряемый класс также должен быть компонентом и его внедрение должно также сопровождаться аннотацией autowired перед сеттером, либо прописано в конфигах

// bean code

@Autowired

public setWheel(Wheel wheel) {

…

}

// java config code

тоже самое, что и Property Injection

// xml config code

<bean id = "car" class = "ru.ifmo.cse.Car">

<property name = "wheel" value-ref = "firstWheelBean" />

</bean>

### **3. Интерфейс на React, формирующий две страницы с разными URL: Главную (/home) и Новости (/news). Переход между страницами должен осуществляться посредством гиперссылок.**

export function App(props) {

return (

<BrowserRouter>

<Routes>

<Route path="/home" element={<div><h1>Home</h1> <a href='/news' > перейти</a></div>} />

<Route path="/news" element={<div><h1>news</h1> <a href='/home' > вернуться</a></div>} />

</Routes>

</BrowserRouter>

);

}

## **Билет 6**

### **1. Структура JSF приложения**

JSF - Фреймворк для Web-приложений на Java, предназначенный для описания пользовательского интерфейса с помощью компонентов, находящихся на сервере. . Он абстрагирует программиста от работы с http протоколом напрямую.

Основан на использовании компонентов.

Фреймворк реализует шаблон MVC.

Структура jsf:

- JSP или XHTML - страницы содержащие компоненты GUI.  
 Элемент (компонент) страницы формирует из себя по сути тэг.  
 JSP или XHTML представляют из себя обычный HTML , но со своими тэгами. Т.е. помимо дефолтных body, input, table и тд, есть еще тэги для кнопок слайдеров и тд.  
 Для этого в стандартной структуре jsf есть:

- Библиотека тэгов - они описывают эти дополнительные тэги jsf.

- Управляемые бины - бины управляемые рантаймом jsf (контейнером), чем является faces Servlet.

- Дополнительные объекты (компоненты, конверторы и валидаторы)

- Дополнительные тэги

- Конфигурация - faces-config.xml (опционально)

- Дескриптор развертывания - как и для любого веб приложения

- JSF фреймворк реализует MVC модель. Он четко разделяет представление от бизнес-логики, в качестве Model выступают написанные нами managed bean’ы, в качестве Controller’а — FacesServlet, который реализован за нас. В качестве View — JSP страница, либо xhtml файл, написанный на xml и позволяющий расширять верстку, добавляя кастомные теги и сторонние библиотеки.

### **2. Spring MVC: особенности, интеграция в Spring**

Фреймворк Spring MVC обеспечивает архитектуру паттерна MVC при помощи слабо связанных готовых компонентов.

1) Model (Модель) инкапсулирует (объединяет) данные приложения, в целом они будут состоять из POJO («Старых добрых Java-объектов», или бинов).● Хранит данные, необходимые для формирования представления. ● Сами по себе эти данные – обычные POJO. ● В общем случае, реализует интерфейс org.springframework.ui.Model.● Есть «упрощённая» реализация,представляющая из себя Map – org.springframework.ui.ModelMap.

2) View (Представление) отвечает за отображение данных фрэймворк не специфицирует жестко технологию на которой будет построено представление. по умолчанию JSP. ● Фреймворк не специфицирует жёстко технологию, на которой должно быть построено представление. ● Вариант «по-умолчанию» – JSP. ● Можно использовать Thymeleaf, FreeMarker, Velocity etc. ● Можно реализовать представление вне контекста Spring – целиком на JS.

3) Controller (Контроллер) обрабатывает запрос пользователя, создаёт соответствующую Модель и передаёт её для отображения на View. ● Класс, который связывает модель с представлением, управляет состоянием модели. ● Помечается аннотацией @Controller. ● Класс или его методы могут быть помечены аннотациями, «привязывающими» их к определённым методам HTTP или URL.

● Spring Web MVC – “базовый” фреймворк в составе Spring для разработки веб-приложений.

● Основан на паттерне MVC (внезапно!)

● Back-end; универсальный, удобен для разработки REST API.

● На клиентской стороне интегрируется с популярными JS-фреймворками.

● Удобно интегрируется с Thymeleaf.

### **3. Написать CDI Bean калькулятор, поддерживающий 4 базовые операции для целых чисел**

@Named(value=”calc”)

@ApplicationScoped

public class Calc implements Serializeable{

public int add(int a, int b) { return a + b; }

public int sub(int a, int b) { return a - b; }

public int mul(int a, int b) { return a \* b; }

public int div(int a, int b) { return a / b; }

}

## **Билет 7 +**

### **1. Spring MVC: обработка запросов, DispatcherServlet**

Вся логика работы Spring MVC построена вокруг DispatcherServlet, который принимает и обрабатывает все HTTP-запросы и ответы на них. При получении запроса, происходит следующая цепочка событий:

1. DispatcherServlet обращается к интерфейсу HandlerMapping, который определяет какой контроллер должен быть вызван ()

2. Контроллер принимает запрос и вызывает соответствующий служебный метод (GET, POST), который возвращает в диспатчер имя View

3. При помощи ViewResolver диспатчер определяет, какой View надо использовать на основании полученного имени

4. После того, как View создан, диспатчер отправляет данные модели в виде атрибутов, которые уже в конечном итоге отображаются в браузере

Dispatcher Servlet

● Обрабатывает все запросы и формирует ответы на них.

● Связывает между собой все элементы архитектуры Spring MVC.

● Обычный сервлет – конфигурируется в web.xml.

### **2. Single Page Application(SPA): преимущества, недостатки**

● Single Page Application – концепция, в соответствии с которой всё приложение «формально» – одна HTML-страница. За фактическую навигацию отвечает клиентский JavaScript – он «подменяет» URL и синхронизирует состояние с сервером.

Это веб-приложение, использующее единственный HTML-документ как оболочку для всех веб-страниц и динамически подгружает HTML, CSS, JS код, обычно посредством AJAX. За навигацию отвечает JS

Клиент и сервер реализуются независимо и взаимодействуют по REST(обычно JSON)

Преимущества:

1. легкость создания из-за огромного количества готовых библиотек и фреймворков.

2. Простое кэширование данных, так как они загружаются в один запрос.

3. Скорость работы, потому что основная часть ресурсов уже загружена, а на страничку подгружаются только необходимые данные, что и экономит время.

Недостатки:

1. “Тяжелые” клиентские фреймворки

2. Без JS невозможно пользоваться полным функционалом приложения

3. Недоступна SEO оптимизация

Основные идеи:

• Все веб приложение - единый веб документ

• Обновление документа при помощи AJAX

• Роутинг через Browser History API

• Оффлайн режим

• При этом проблемы c SEO, аналитикой и security

### **3. Интерфейс JSF (xhtml страница + CDI), реализующий ввод паспортных данных (серия, номер, дата, место выдачи)**

@ManagedBean

@SessionScoped

@Setter

@Getter

public class PassportInputBean {

private String series;

private String number;

private Date date;

private String place;

@ManagedProperty(value = "#{repoBean}")

private PassportRepo repo;

public void storePassportData() {

repo.store(series, number, date, place);

}

}

<!DOCTYPE html>

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

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

xmlns:f="http://xmlns.jcp.org/jsf/core">

<head>

<title>Форма для ввода паспортных данных</title>

</head>

<body>

<h:form>

Введите серию

<h:inputText type="text" value="#{passportInputBean.series}"

required="true"/>

Введите номер

<h:inputText type="text" value="#{passportInputBean.number}"/>

Введите дату

<h:inputText value="#{passportInputBean.date}">

<f:convertDateTime pattern="yyyy-MM-dd"/>

</h:inputText>

Введите место рождения

<h:inputText type="text" value="#{passportInputBean.place}"/>

<h:commandButton value="Отправить" action="#{passportInputBean.storePassportData()}"/>

</h:form>

</body>

</html>

## **Билет 8 +**

### **1. Технология RMI. Использование RMI в Java EE**

RMI — Java API, позволяющий вызывать методы удалённых объектов.

Система Java Remote Method Invocation (RMI) позволяет объекту, запущенному на одной виртуальной машине Java, вызывать методы объекта, запущенного на другой виртуальной машине Java. Работает поверх TCP. В общем случае, объекты передаются по значению (копии), следовательно, передаваемые объекты должны быть Serializable.

Использование:

1) Регистрируем серверный объект RMI Registry

Далее создается заглушка, реализующая тот же интерфейс что и серверный объект, отправляется клиенту и притворяется, что все методы есть.

2) При вызове происходит поиск объекта сервера

3) Клиенту прилетает ответ

4) Происходит обмен данными

Клиент работает с сервером через интерфейс с серверной заглушкой, думая, что это сам серверный объект.

### **2. Управление состоянием в React. Flux & Redux**

1) Flux — архитектура для создания приложений на React, в которой описывается, как хранить, изменять и отображать глобальные данные (состояние).  
Основные концепции:  
Dispatcher принимает события от представления (например от кнопок интерфейса) и отправляет их на обработку хранилищу данных (Store).  
Только Store знает, как менять данные. Напрямую из React-компонента их изменить нельзя. После изменения данных Store посылает события представлению, и оно перерисовывается.  
2) Redux — небольшая библиотека, реализующая упрощенный паттерн Flux. Redux предоставляет единое хранилище состояния. Redux - хранилище для всего нашего приложения и мы можем из любого компонента получить состояние переменной, который находится в общем хранилище и подписаться на ее изменение, либо кинуть экшен на ее изменение. Ключевой его особенностью является неизменяемость — все изменения в него вносятся посредством создания нового объекта состояния.

В “правильном” React-приложении с использованием Redux у компонентов нет state’а

Есть store — синглтон, хранилище состояние всего приложения. Изменения состояния производятся при помощи чистых функций (reducers). Они принимают на вход state и действие(action) и возвращают либо неизмененный state либо копию

- Store

Объект, который содержит состояние приложения и является уникальным.

Является неизменяемым - обновляется путем создания нового объекта. (удаляем прошлое значение)

- Reducer

● Чистая функция, её выходное значение зависит только от входных

● Необходимо возвращать новое состояние, состояние должно быть неизменяемо

- Action

Некоторое событие, которое должно привести к изменению состояния (созданию нового объекта состояния), будь то ответ от сервера или нажатие на кнопку в интерфейсе.

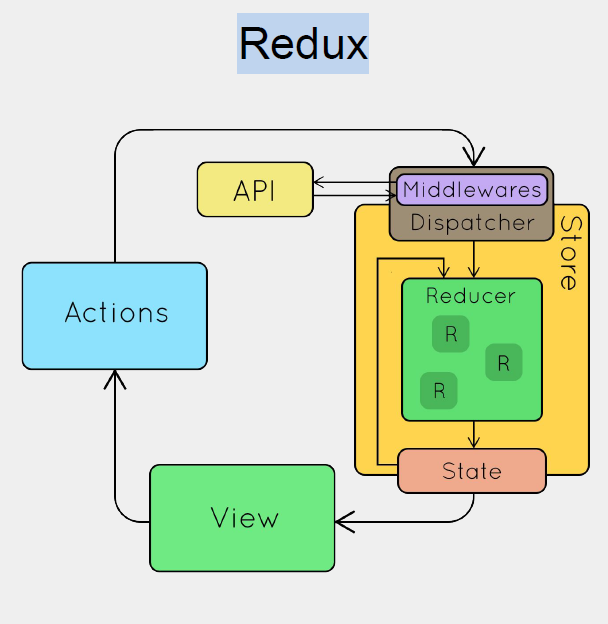
С точки зрения библиотеки — любой JS объект, у которого есть идентификатор типа события (обычно строковое свойство type) и некоторая полезная нагрузка.

- Dispatch

Функция redux, которая генерирует события.

- Middleware

Усилители. Главная их суть - принять входные данные действия, обработать и отправить дальше. Часто используются для работы с api



### **3. Привести фрагмент кода управляемого бина, увеличивающего на 1 значение, отображаемое на кнопке при каждом клике по ней:**

@ManagedBean

@ApplicationScoped

public class MyBean implements Serializable {

private int value = 0;

public void increment() {

value++;

}

public int getValue() {

return value;

}

public void setValue(int value) {

this.value = value;

}

}

<h:commandButton action= "#{myBean.increment}" value = "#{bean.value}"/>

## **Билет 9 +**

### **1. JNDI. JNDI в Java EE. Способы взаимодействия с JNDI. Их преимущества и недостатки.**

JNDI — API для доступа к объектам и ресурсам по их именам. Организовано в виде службы имен и каталогов. Образует иерархическую структуру.

JNDI — это набор Java API, организованный в виде службы каталогов, который позволяет Java-клиентам открывать и просматривать данные и объекты по их именам  
Чаще всего JNDI используется в enterprise-разработке. Главный юзкейс — настройка доступа к базе данных. Приложение знает только JNDI-имя, например “java:comp/env/Db” а сами детали подключения описываются администратором в веб контейнере.

JNDI API позволяет осуществлять удобный поиск по каталогам, давая удобный доступ к бинам, данным для входа в базы данных, локализованные ресурсы и т.д.

Юзать можно вот так, например:

Name name = new CompositeName("java:comp/env/jdbc");  
JNDI поддерживает разные реализации сервиса служб имен и каталогов. Некоторые из них: DNS, RMI, LDAP, COBRA.

Изображение выглядит как текст, снимок экрана, Шрифт, линия

Автоматически созданное описание  
Преимущества JNDI:  
 - удобство настройки приложения  
 - пароли к бд лежат отдельно от приложения, что безопаснее  
 - при изменении бд (например от тестовой к проду) не нужно пересобирать приложение  
Недостатки:  
 - зависимость от контейнера  
 - при использовании старой версии log4j есть уязвимость (log4shell) основанная на jndi

2 варианта использования:

-CDI аннотации, работает только в managed компонентах

-прямой вызов API, работает везде. new InitialContext().lookup(“res”);

### **2. React. Особенности. Архитектура**

React — JS библиотека для разработки пользовательского интерфейса (в основном SPA). Позволяет создавать свои собственные компоненты, с пропсами (атрибутами) и стейтом (данными). Компоненты рендерятся в HTML.

React — библиотека для разработки графический интерфейсов (не только веб). Использует компонентный подход. В нем существует корневой элемент App, в который вставляются новый компоненты. Структура древовидная. Так же реакт реализует подход одностраничного приложения, то есть всего есть 1 страница, которая перерисовывается JS’ом.

Написание происходит в своем расширении .jsx или .tsx  
Особенности:  
- передача данных от родителя к детям. One way dataflow  
- виртуальный DOM (свое собственное дерево компонентов, параллельно с DOM)  
- JSX - это JS с HTML внутри

- При изменении state происходит ререндер компонента с обновлением вложенных компонентов/тэгов.  
React отвечает за представление интерфейса и не навязывает какую-то определенную архитектуру. При написании сложных приложений, работающих с большим количеством данных, часто применяется архитектура Flux и библиотека Redux.

Для привычной навигации по ссылкам существует React Router. Компоненты умные, имеют Props, который позволяет одному компоненту выглядеть по-разному на разных Route’ах, например.

### **3. Сделать бин который показывает время в минутах со старта сервера**

@Named(“serverTimer”)

@ApplicaitonScoped

public class ServerTimer {

private long start;

public ServerTimer() {

start = System.currentTimeMillis();

}

public long getTime() {

return (System.currentTimeMillis() - start) / 60000;

}

}

## **Билет 10 +**

### **1. Платформы Java. Сходства и различия.**

Существует 3 самых известных API: JME, JSE, JEE

JME-(Java Micro Edition) представляет из себя API и минимально требовательную VM для разработки и старта приложения на смартфоне/планшете. Основана на более ранней версии JSE, поэтому некоторые функции (напр. дженерики) не работают

JSE-(Java Standard Edition) занимается обеспечением основными стандартными функциями самой Java, и она определяет абсолютно все: базовые типы и объекты языка, классы более высокого уровня, производительность приложения в сети и обеспечение защищенности. Включает в себя пакеты: java.util,java.math,java.io,java.nio,java.lang и тд.

JEE-(Java Enterprise Edition) для разработки Enterprise приложений. Она строится на основе платформы JSE, а еще дает возможность разработки более крупно масштабируемых , сложно уровневых и безопасных программ. Содержит: WebSocket, JSF, Unified EL, API для веб-служб RESTful, DI, EJB, JPA, и Java Transaction API.  
Все платформы Java поддерживают полный функционал языка Java и отличаются лишь наличием или отсутствием определенных API.

### **2. Двухфазные и трехфазные конструкторы в Spring и Java EE**

Двухфазовые конструкторы в Spring: обычный конструктор + метод с аннотацией @PostConstruct. Сначала вызовется обычный конструктор, а затем помеченный метод. На момент его вызова все зависимости будут обработаны и доступны. Аннотация является частью Java EE и работает в Spring. Это иногда удобно, так как при вызове конструктора Autowired и Inject еще не подгрузили зависимости, а вот в методе уже все есть.  
Трехфазовый конструктор(третья фаза изменения объекта до его использования) в Spring : сначала был нативный конструктор, потом обработались зависимости, уже с обработанными зависимостями вызвалось конструирование объекта, а потом добавились срезы(аспекты).

Аспекты(срезы)- это “вкрапления”, которые позволяют добавить поведение до/после вызова оригинального метода через Proxy. Такой конструктор имеет свой скоуп. 3) Реализаций его нет, нужно ручками писать. То есть мы определяем поведение объекта до вызова метода. Но для этого нужно наследоваться от BeanPostProcessor’а, и там уже прописывать логику.

### **3. Написать страницу JSF, которая бы выводила сначала 10 простых чисел, а затем с помощью Ajax запроса ещё 10.**

@ApplicationScoped

@Named

public class PrimeNumber {

@Getter // Lombok

private final List<Long> primes = new ArrayList<>();

public PrimeNumber() {

nextPrimes();

}

public void nextPrimes() {

// Добавляет в primes следующие 10 простых чисел

}

}

<!DOCTYPE html>

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

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

xmlns:f="http://xmlns.jcp.org/jsf/core">

<h:head>

<title>Показать простые числа</title>

</h:head>

<h:body>

<h:form>

<h:dataTable id="table" value="#{primeNumber.primes}" var="prime">

<h:column>

<h:outputText value="#{prime}"/>

</h:column>

</h:dataTable>

<h:commandButton value="Следующие 10" action="#{primeNumber.nextPrimes}">

<f:ajax execute="@form" render="table"/>

</h:commandButton>

</h:form>

</h:body>

</html>

## **Билет 11**

### **1. Java EE CDI Beans стереотипы**

Stereotype — аннотация, включающая в себя много аннотаций. Мы можем создать свой стереотип и использовать его:

@ApplicaitonScoped

@Named

@Secure

public @interface myStereotype() {}

Также существуют стандартные стереотипы, например @Model (@RequestScoped + @Named)

При применении аннотации @MyStereotype будут включатся все перечисленные аннотации. Один бин может использовать несколько стереотипов.

Если у вас будут разные scope в стереотипах, то у вас не скомпилится, либо вам надо будет указать скоуп прямо перед бином. По поводу имен бинов ещё круче: в стереотипах нельзя задавать имена бинов

### **2. Разметка страницы в React-приложениях. JSX**

Разметка страницы в React-приложениях обычно выполняется с использованием JSX (JavaScript XML), который предоставляет удобный способ описания пользовательского интерфейса. JSX похож на HTML, но использует синтаксис JavaScript для описания компонентов.

Точкой входа (корнем) являет index.js который определяет компоненты. Каждый компонент включает в себя другие компоненты.

В React приложениях разметка пишется в JSX файлах. JSX — надстройка над JS, которая позволяет вкраплять HTML-синтаксис в код. Можно использовать стандартные HTML элементы (такие как div, span, h1, input) так и кастомные React компоненты.  
JSX код: <div className=”foo”>text</div> компилируется в вызов функции React.createElement(“div”, { className: “foo”}, “text”)

JSX:

● Расширение языка JavaScript

● Cахар для React.createElement(component, props, ...children)

● Компилируется Babel’ом в JS

● визуально близок к HTML

Правила JSX

Возвращает один корневой элемент

Чтобы вернуть несколько элементов из компонента, оберните их одним родительским тегом (например <div> или вместо них можно написать <> и </> (называется фрагмент))

JSX требует, чтобы теги были явно закрыты: самозакрывающиеся теги, такие как <img> должны стать <img />, а оберточные теги типа <li>oranges должны быть записаны как <li>oranges</li>. В React многие атрибуты записываются в camelCase.

- JSX — переиспользование компонентов

class NameForm extends React.Component {

render() {

return (

<div> smth </div>

);

}}

class ComponentThatUseAnotherComponent extends

React.Component {

render() {

return (

<div>

We use component from previous slide

<NameForm/>

</div>

);

}}

Важно: Пользовательские компоненты должны быть написаны с большой буквы

В фигурных скобках ‘{}’ JS выражения. Т.к. это выражения никаких if, for, объявлений и прочего.

render() {

let name = ‘Vasya’;

return (

<div>

<h3>Name { this.name}</h3>

<TodoList items={ this.state.items} />

</div>

);

}

Условный рендер в JSX

class Greeting extends React.Component {

render() {

const isLoggedIn = props.isLoggedIn;

if (isLoggedIn) { return <UserGreeting />; }

return <GuestGreeting />;

}}

Циклы в JSX

function Item(props) { return <li>{props.message}</li>; }

function TodoList() {

const todos = ['finish doc', 'submit pr'];

return (

<ul>

{todos.map((message) => <Item key={message}

message={message} />

)}

</ul>

);

}

### **3. JSF Manager Bean, после инициализации HTTP-сессии формирующий коллекцию с содержимым таблицы Н\_УЧЕБНЫЕ\_ПЛАНЫ. Для доступа к БД необходимо использовать JDBC-ресурс jdbc/OrbisPool.**

@Named("myBean")

@SessionScoped

class MyBean {

private List<String> plans;

private List<String> getPlans() {

return plans;

}

@Resource(name="jdbc/OrbisPool",type=DataSource.class)

private DataSource dataSource;

@PostConstruct

private void loadPlans() {

try (var conn = dataSource.getConnection()) {

var stmt = conn.createStatement();

var rs = stmt.executeQuery("SELECT name FROM plans");

plans = new ArrayList<>();

while (rs.next()) {

plans.add(rs.getString("name"));

}

}

}

}

## **Билет 12**

### **1. Location Transparency в Java EE**

Принцип Location Transparency означает, что благодаря CDI мы можем добиться того, что нам станет не важно, где физически расположен вызываемый компонент (локально, удалённо) - за его вызов отвечает контейнер. Если мы, к примеру, с JNDI ищем DataSource, нам не будет важно, где находится объект, ссылку на который мы получим. Таким образом, мы можем одинаково обращаться как к локальному, так и к удалённому объекту. Его получением занимается сервер приложений.

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.

В Java EE в первую очередь реализуется через JNDI — API для предоставления доступа к объектам по имени, нежели по их физической локации.

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 расположения ресурсов. Не важно где они находятся, обращение к ним одинаковое.

Контейнер будет таскать объекты с помощью RMI.

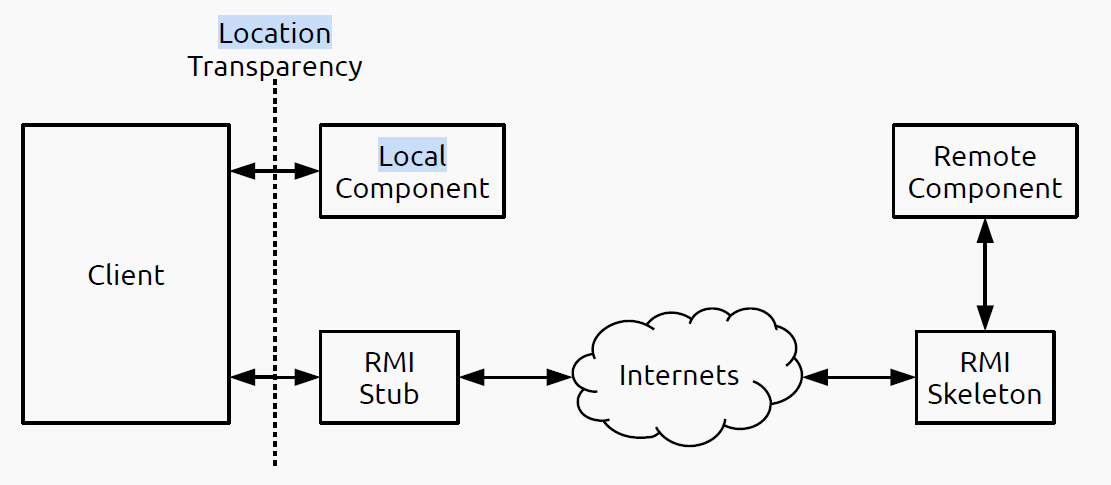
Для клиента создается видимость целостности приложения, как будто оно не расположено по разным серверам с разными JVM, то есть для использования Remote EJB ему не надо делать дополнительных движений.

JNDI: JEE использует JNDI для обеспечения прозрачного доступа к удаленным компонентам. JNDI позволяет приложениям получать доступ к объектам по их именам, а не по конкретным адресам. Это дает возможность динамического обнаружения и использования удаленных ресурсов без явного указания их расположения.

RMI: JEE использует RMI для вызова методов удаленных объектов. RMI позволяет вызывать методы объектов, которые находятся на удаленных машинах, как если бы они находились локально. Это также обеспечивает прозрачность местоположения, поскольку клиентский код не заботится о том, где находится удаленный объект.

EJB: EJB компоненты в JEE могут быть развернуты как локальные или удаленные. Благодаря этому клиентский код может использовать эти компоненты, не зная, где они находятся.

Контексты и Dependency Injection: Введение Dependency Injection (DI) в JEE также способствует прозрачности местоположения. С помощью DI компоненты приложения могут быть внедрены в другие компоненты, не заботясь о том, где эти компоненты находятся.



Stub: Stub (заглушка) - это локальный объект, который представляет удаленный объект. Он служит как клиентский прокси для взаимодействия с удаленным объектом. Когда клиент вызывает метод на stub, он делегирует этот вызов удаленному объекту по сети. Stub выполняет необходимую сериализацию параметров вызова и отправляет их по сети на удаленный сервер. После выполнения вызванного метода удаленным объектом, stub получает результат и передает его обратно клиентскому коду. Фактически, stub обеспечивает локальное представление удаленного объекта.

Skeleton - это серверный объект, который получает запросы от клиентов через сеть и делегирует их на удаленные объекты для выполнения. Когда клиент отправляет запрос на удаленный вызов метода, skeleton принимает этот запрос, извлекает параметры, необходимые для вызова метода на удаленном объекте, и перенаправляет запрос на нужный объект для обработки. После выполнения метода удаленным объектом, скелет форматирует результат и отправляет его обратно клиенту через сеть.

### **2. Spring MVC часть представления**

Вся логика работы Spring MVC построена вокруг DispatcherServlet, который принимает и обрабатывает все HTTP-запросы и ответы на них. При получении запроса, происходит следующая цепочка событий:

1. DispatcherServlet обращается к интерфейсу HandlerMapping, который определяет какой контроллер должен быть вызван ()

2. Контроллер принимает запрос и вызывает соответствующий служебный метод (GET, POST), который возвращает в диспатчер имя View

3. При помощи ViewResolver диспатчер определяет, какой View надо использовать на основании полученного имени

4. После того, как View создан, диспатчер отправляет данные модели в виде атрибутов, которые уже в конечном итоге отображаются в браузере

Представление отвечает за то, как будут визуализироваться данные в браузере пользователя.  
За поиск представления по имени отвечает интерфейс ViewResolver.

Фреймворк не специфицирует жестко технологию на которой будет построено представление

по умолчанию JSP: можно использовать Thymeleaf, freemaker, velocity и тд., можно реализовать представление вне контекста Спринг - целиком на JS

ViewResolver - интерфейс, при помощи которого DispatcherServlet определяет какое представление нужно использовать на основании имени.

● Представление в Spring Web MVC может быть построено на разных технологиях.

● С каждым представлением сопоставляется его символическое имя.

● Преобразованием символических имён в ссылки на конкретные представления занимается специальный класс,

реализующий интерфейс org.springframework.web.servlet.ViewResolver.

● Существует много реализаций ViewResolver для разных технологий построения представления.

● В одном приложении можно использовать несколько ViewResolver'ов.

### **3. Написать managed bean и задать ему scope такой же как у бина otherBean**

@ManagedBean

@ApplicationScoped

public class OtherBean {

@ManagedProperty(value="#{myBean}")

@Getter // Lombok

private MyBean myBean;

}

@ManagedBean

@NoneScoped

@Named

public class MyBean {}

## **Билет 13 +**

### **1. Валидаторы в JSF. Создание, назначение и тд.**

Валидаторы в JSF — реализации интерфейса Validator. Метод validate принимает FacesContext, UiComponent и значение (Object), которое необходимо валидировать.

● Осуществляется перед обновлением значения компонента на уровне модели.

● Класс, осуществляющий валидацию, должен реализовывать интерфейс javax.faces.validator.Validator.

● Существуют стандартные валидаторы для основных типов данных. DoubleRangeValidator, LengthValidator, RegexValidator, RequiredValidator.

● Можно создавать собственные валидаторы c помощью аннотации @FacesValidator

Способы:

-С помощью параметров компонента: required="true"

-С помощью вложенного тега <f:validatorvalidatorId=”com.example.MyValidator”/>

-С помощью логики на уровне управлямого бина

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, реализующийинтерфейс Validator (javax.faces.validator.Validator).

- Реализовать метод validate().

- Зарегистрировать валидатор в файле faces-config.xml или аннотацией@FacesValidator

- Использовать тег на страницах JSP.

### **2. Реализация контроллера в Spring Web MVC**

Контроллер – специальный класс, помеченный аннотацией @Controller (или RestController).

● связывает модель с представлением, управляет состоянием модели.

● Помечается аннотацией @Controller.

● Класс или его методы могут быть помечены

аннотациями, «привязывающими» их к

определённым методам HTTP или URL.

Задача контроллера - перехватывать входящие запросы, паковать данные в нужный формат, отправлять эти данные нужной модели, а затем ответ от модели передать обратно в DispatcherServlet. На методы контроллера вешаются разные аннотации, такие как @GetMapping, @PostMapping (это для путей запроса) и аннотации для получения данных из запроса, такие как @PathVariable или @ResourcesVarible.

Пример:

@Controller

public class HelloController {

@RequestMapping(value = "/hello",

method = RequestMethod.GET)

public String printHello(ModelMap model) {

model.addAttribute("message",

"Hello Spring MVC Framework!");

return "hello";

}

}

### 

## **Билет 14 +**

### **1. Фаза формирования представления**

● JSF Runtime формирует представление (начиная с UIViewRoot):

● Создаются объекты компонентов.

● Назначаются слушатели событий, конвертеры и валидаторы.

● Все элементы представления помещаются в FacesContext.

● Если это первый запрос пользователя к странице JSF, то формируется пустое представление.

● Если это запрос к уже существующей странице, то JSF Runtime синхронизирует состояние компонентов представления с клиентом.

В фазе **Restore View** фреймворк выполняет следующие действия :

* Проверка FacesContext текущего запроса. Если он содержит UIViewRoot, то
  + Установить значение locale для UIViewRoot, извлекаемую из метода ExternalContext.html#getRequestLocale() текущего запроса.
  + Для каждого компонента дерева проверить значение ValueBinding связи "binding". Если связь определена, то у ValueBinding вызвать метод setValue().
  + Не предпринимать дальнейших действий на этой стадии.
* Определение идентификатора текущего запроса вида (view identifier) :
  + если используется маппинг с префиксом (например, "/faces/\*"), то в качестве viewId использовать значение, определенное на месте символа '\*';
  + если используется маппинг с суффиксом (например, "\*.faces"), то в качестве viewId назначается путь, по которому был сделан запрос;
  + Если идентификатор определить не удаётся, то вызвать исключение.
* Вызов метода ViewHandler#restoreView() с передачей в качестве параметров FacesContext и идентификатора вида. Результат выполнения (возможно) UIViewRoot.
  + если restoreView() возвращает null, то последовательно вызываются ViewHandler#createView() и FacesContext#renderResponse();
  + если запрос не содержит POST-данных или параметров (query parameters), то вызывается FacesContext#renderResponse().
* Сохранение созданного или восстановленного UIViewRoot в FacesContext'е.

В конце данной фазы свойство viewRoot у FacesContext'а текущего запроса будет содержать сохранённое состояние вида, которое было в предыдущем запросе или новый вид, созданный с помощью ViewHandler#createView() для указанного идентификатора вида.

### **2. Spring framework. Отличия от Java EE**

Java EE — это, грубо говоря, конструктор, она модульная, можно делать свои сборки, подключать и отключать совсем маленькие модули. В ней есть множество реализаций представлений, бинов и т.д. что делает ее идеальной для разработки монолитного масштабируемого приложения.

Spring — также разделен на модули, но эти модули довольно крупные и скорее удобно дополняют друг-друга чем живут обособленно. Спринг ставит более жесткие рамки, диктуя как писать приложении, и в каких-то случаях это очень важно. Этот фреймворк подходит для небольших веб приложений, либо для микросервисной архитектуры. Тот же ajax, который в JavaEE пишется 1 строчкой (благодаря JSF), в спринге нужно писать руками.

-Spring медленнее

-контейнер в JAVA EE включает в себя приложение, а Spring наоборот, включает в себя контейнер

-JAVA EE - спецификация, Spring - фрэймворк

- концепция JAVA EE - разделение обязанностей между контейнером и компонентом, концепция Spring - IOC, CDI

### **3. RestController, который реализует перевод градусов Цельсия в Фаренгейты и обратно**

@RestController

class TempController {

@GetMapping("/convert/c/f")

double celsiusToFahrenheit(@RequestParam double celsius) {

return celsius \* 1.8 + 32.0;

}

@GetMapping("/convert/f/c")

double fahrenheitToCelsius(@RequestParam double fahrenheit) {

return (fahrenheit - 32.0) / 1.8;

}

}

## **Билет 15 +**

### **1. Java EE CDI Beans прерывание жизненного цикла (Interception)**

В спецификации CDI предусмотрен механизм «перехватчиков», который предоставляет возможность добавить к методу бина предобработку и постобработку.

Interceptorы - классы, реагирующие на определённые события жизненного цикла бинов

Для связи метода с интерсептором необходимо создать кастомную аннотацию, которая дополнительно помечается аннотацией @InterceptorBinding.  
Как внедрить дополнительную логику до и после вызываемого метода:

1. Создаем кастомную аннотацию, над которой надо указать @InterceptorBinding

2. Далее создать класс, который и будет в роли Интерсептора, повесить на него созданную нами аннотацию + @Interceptor

3. Создать метод для обработки, с параметром InvocationContext ctx пометить его аннотацией @AroundInvoke. В самом методе, чтобы вызвать метод пишем ctx.proceed();. До этого или после мы можем описать дополнительную логику.

4. Теперь осталось добавить нашу кастомную аннотацию методу, чью логику мы хотим расширить (если повесить аннотацию на класс, то все его методы будут прерываться)

### **2. Компоненты React. State & props. "Умные" и "Глупые" компоненты**

Компоненты позволяют разбить интерфейс на независимые части, про которые легко думать в отдельности. Их можно складывать вместе и использовать несколько раз. Во многом компоненты ведут себя как обычные функции JavaScript. Они принимают произвольные входные данные (так называемые «пропсы») и возвращают React-элементы, описывающие, что мы хотим увидеть на экране.

Чтобы отрендерить элемент используем функцию ReactDom.render(element, document.getElementById(‘root’); Создать компонент можно 2-мя способами: Создать класс, расширяющий React.Component, либо создать функцию, возвращающую View.

Жизненный цикл

- инициализация — создание экземпляра класса компонента

- монтирование — компонент вставляется в DOM

- обновление — компонент обновляется новыми данными через состояние или свойства

- размонтирование — компонент удаляется из DOM

Пропсы (сокращённо от «properties») и state — это обычные JavaScript-объекты которые содержат инфу влияющую на представление. props передаётся в компонент (как параметры функции), в то время как state находится внутри компонента (по аналогии с переменными, которые объявлены внутри функции).

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

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

"Умные" компоненты хранят в себе состояние и могут меняться в зависимости от него. Они управляют простыми компонентами, делают запросы на сервер и многое другое.

"Глупые" компоненты не выполняют никаких сверхсложных задач. Все их действия просты и однообразны, они всего лишь выводят данные, принимаемые ими от свойств (пропсов).

Глупые компоненты:

1. не зависят от остальнойчасти 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

### **3. Написать JSF страницу и многострочным полем, в которое можно вводить только строчные символы латиницы.**

<!DOCTYPE html>

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

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

xmlns:f="http://xmlns.jcp.org/jsf/core">

<h:head>

<title>латиница</title>

</h:head>

<h:body>

<h:inputTextArea>

<f:validateRegex pattern=”[a-z]\*”/>

</h:inputTextArea>

</h:body>

</html>

## **Билет 16**

### **REST контроллеры в спринге. Сериализация и десериализация**

@RestController указывает, что класс является контроллером, где метод @RequestMapping предполагает @ResponseBody по умолчанию (т. е. REST API, который возвращает данные в виде JSON или XML). Для обработки запросов нужно написать класс со спец. аннотацией @RestController (@Controller + @ResponseBody), в котором будет реализована логика обработки клиентских запросов. @RestController — говорит спрингу, что данный класс является REST контроллером. Т.е. в данном классе будет реализована логика обработки клиентских запросов.

В Accept Header можно передать тип данных в котором нужно получить ответ. Программисту ничего делать не надо.

Spring сам не умеет в сериализацию /маршалинг. Маршаллинг - это передача сущности из одного контекста в другой

● Сериализация / маршалинг реализуются сторонними библиотеками.

● HttpMessageConverter – адаптер для сторонних библиотек.

● Содержит 4 метода – canRead(MediaType), canWrite(MediaType), read(Object, InputStream, MediaType) и write(Object, OutputStream, MediaType).

Специальными аннотациями @Get/Post/Put/DeleteMapping помечаются методы для обработки http запросов. В аргументах аннотации можно указать path, по которому можно обратиться к данному методу. @RequestBody помечается десериализованный из запроса объект. DTO (data transfer object).

@RequestBody, - десереализует автоматически жававский объект из JSON

@ResponseBody - сереализует в JSON для передачи клиенту, указывает, что тип результата должен быть записан прямо в теле ответа в любом указанном вами формате, например JSON или XML.

@RequestParam можно использовать для обработки переменных шаблона в сопоставлении URI запроса и задавать их в качестве параметров метода, можно вытащить часть GET параметра

@PathVariable используется для привязки параметра метода к переменной path, можно использовать для извлечения динамической части URL, когда данные приходят как часть URL

Десериализация устроена следующим образом: если присутствует тело запроса, прописан заголовок Content-Type, и обработчик запроса принимает аргумент, помеченный аннотацией @RequestBody, Spring автоматически десериализует данные, используя Jackson. Из коробки доступен формат JSON, но можно установить поддержку XML. Помимо тела запроса, данные могут приходить как часть URL: их можно вытащить через @PathVariable, или как часть GET-параметра, используя @RequestParam.  
Сериализация работает похожим образом. Из обработчика возвращается объект, а Spring его автоматически сериализует. Формат выбирается исходя из HTTP заголовка Accept. Формат можно прописать вручную в свойстве produces аннотации @RequestMapping.

### **2. Архитектура Angular приложения. Модули, компоненты, представление, сервисы**

Приложение разбивается на модули, которые могут импортировать друг в друга. Есть набор стандартных модулей Angular. Сторонние библиотеки также могут предоставлять модули.  
В модулях определяются компоненты. Компоненты — строительные блоки интерфейса, кирпичики, инкапсулирующие верстку, стили, и логику приложения, которые можно переиспользовать внутри других компонентов. Любой компонент состоит из TS класса, помеченного декоратором @Component, HTML шаблона, CSS стилей. Компонент вместе с шаблоном образуют представление. Представления образуют иерархию. Существует двухсторонняя связь между классом компонента и представлением — при изменении данных в компоненте обновляется представление и наоборот.  
Задачи приложения, которые не касаются представления, выносятся в сервисы. Это, к примеру, загрузка данных с сервера, валидация данных, фоновые процессы, логирование. Angular поддерживает DI: сервисы внедряются в компоненты. Для этого используется аннотация @Injectable.

**Конфигурация проекта**

• Удобный и гибкий CLI

• angular.json — скелет приложения

• Очень гибкая конфигурация проекта

• tsconfig.json

• tslint.json

• Karma, Jasmine для тестов

• Кастомизация сборки

**Модули**

• **Модуль** - это класс с декоратором @NgModule(), который служит изолирующей логической объединяющей структурой для компонентов, директив, пайпов и сервисов.

• Каждое приложение обязательно включает в себя корневой модуль (root module) под названием AppModule (файл app.module.ts).

• Могут ссылаться друг на друга (т.е. возможны импорт и экспорт модулей).

Содержит секции:

•declarations -- компоненты, директивы (directives) и фильтры (pipes), содержащиеся в этом модуле.

•exports -- то, что объявлено в этой секции, будет видно и доступно для использования в других модулях.

•imports -- список внешних модулей, содержимое секции exports которых используется в текущем модуле.

•providers -- сервисы, реализованные в этом модуле, видимые в глобальном контексте приложения.

•bootstrap -- главное представление приложения (объявляется только в корневом модуле).

● Компоненты и их шаблоны формируют представления (views).

● Компонент может содержать иерархию представлений (view hierarchy).

● Каждый компонент содержит корневое представление (host view).

● Представление (view) компонента задаётся с помощью шаблона (template).

● Представления часто группируются иерархически.

● Компонент может содержать иерархию представлений (view hierarchy), которая содержит встроенные представления (embedded views) из других компонентов.

**Компонент**

**Компонент** - обособленная часть функционала со своей логикой, HTML-шаблоном и CSS-стилями. ● Каждый компонент -- отдельный класс.

● Контролирует область экрана, называемую *представлением (view)*.

● Angular управляет жизненным циклом компонентов.

За объявление компонента отвечает декоратор @Component()

Основные свойства объекта, который принимает декоратор:

•selector — название компонента;

•template (или templateUrl) — HTML-разметка в виде строки (или путь к HTML- файлу);

•providers — список сервисов, поставляющих данные для компонента;

•styles — массив путей к CSS-файлам, содержащим стили для создаваемого компонента.

Жизненный цикл компонента:

**- OnChanges** - устанавливаются или изменяются значения входных свойств класса компонента;

**- OnInit** - устанавливаются "обычные" свойства; вызывается единожды вслед за первым вызовом OnChanges();

**- DoCheck** - происходит изменения свойства или вызывается какое-либо событие;

**- AfterContentInit** - в шаблон включается контент, заключенный между тегами компонента;

**- AfterContentChecked** - аналогичен DoCheck(), только используется для контента, заключенного между тегами компонента;

**- AfterViewInit** - инициализируются компоненты, которые входят в шаблон текущего компонента;

**- AfterViewChecked** - аналогичен DoCheck(), только используется для дочерних компонентов;

**- OnDestroy** - компонент "умирает", т.е. удаляется из DOM-дерева

**Сервис**

**Сервис** - это класс, который является поставщиком данных. Сервисы инкапсулируют бизнес логику приложения. Сервисы могут предоставлять интерфейс взаимодействия между отдельными не связанными друг с другом компонентами.

● Реализуются в виде отдельных классов в соответствии с принципами ООП.

● Компонент может делегировать какие-либо из своих задач сервисам.

● Доступ компонентов к сервисам реализуется с помощью DI.

**Dependency injection**

• Компоненты могут использовать сервисы с помощью DI.

• Для того, чтобы класс можно было использовать с помощью DI, он должен содержать декоратор @Injectable().

### **3. Конфигурация faces-config, задающая managed bean с именем myBean, которым будет управлять сам программист**

<?xml version="1.0" encoding="UTF-8"?>

<faces-config xmlns="http://xmlns.jcp.org/xml/ns/javaee"

xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"

xsi:schemaLocation="http://xmlns.jcp.org/xml/ns/javaee

http://xmlns.jcp.org/xml/ns/javaee/web-facesconfig\_2\_3.xsd"

version="2.3">

<managed-bean>

<managed-bean-name>myBean</managed-bean-name>

<managed-bean-class>com.example.MyBeanClass</managed-bean-class>

<managed-bean-scope>custom</managed-bean-scope>

</managed-bean>

</faces-config>

## **Билет 17 +**

### **1. JSF: ключевые особенности, преимущества, недостатки**

JSF — фреймворк для создания веб приложений на Java, является частью стандарта Java EE. Он абстрагирует программиста от работы с http протоколом напрямую.  
Особенности:  
 - Компонентно-ориентированная структура. Интерфейс строится из компонентов, которые могут быть вложены друг в друга. Рендерятся в HTML элементы.  
 - Для отображения данных используются JSP или XML-шаблоны (facelets).  
 - Бизнес логика выносится в Java бины

● JSF — фреймворк для разработки веб-приложений.

● Входит в состав платформы Java EE.  
Преимущества:  
 - разделение бизнес логики от представления (реализует MVC где Model - Managed Beans, View - JSP или XHTML страница, а Controller - FacesServlet)

- Управление обменом данными на уровне компонент.

- программисту нужно писать меньше JS кода  
- простота реализации AJAX

- работа с событиями на стороне сервера

- расширяемость(доп наборы компонент, можно определять свои)

- поддержка в IDE  
Недостатки  
 - Плохо масштабируется. сложно реализовывать не предусмотренную авторами функциональность и компоненты  
 - Не подходит для высокопроизводительных приложений  
 - learning curve

### **2. CDI-бины: что такое и зачем нужны, когда есть EJB и Managed Beans**

CDI бины - специальные бины, которые позволяют разработчику использовать концепцию внедрения зависимостей. CDI даёт возможность управлять bean-компонентами.

● Универсальные компоненты уровня бизнес-логики.

● Появились в Java EE 6, копируют концепции, реализованные в Spring.

● Общая идея – «отвязаться» от конкретного фреймворка при создании бизнес-логики внутри приложения.

● В большинстве случаев их можно использовать вместо JSF Managed Beans и EJB.

● По реализации очень похожи на JSF Managed Beans.

В отличие от MB, CDI бины намного мощнее и гибче, они могут использовать перехватчики, стереотипы, декораторы и многое другое, а также смешиваться с другими бинами. EJB же обладают некоторыми особенностями, недоступными для CDI (например, транзакционные функции, таймеры, асинхронность, удаленность). Однако, в целом, EJB и CDI схожи, и их можно даже инжектить друг в друга.

### **3. Angular компонент, который позволяет поделиться чем-то в VK, Twitter, Facebook (API для соцсетей можно описать текстом). API принимает логин и пароль, а также сообщение, которое будет опубликовано на личной странице после прохождения авторизации.**

import { Component} from '@angular/core';

import {HttpClient} from '@angular/common/http';

@Component({

selector: 'my-app',

template: <div class="form-group">

<label>Login</label>

<input class="form-control" name="username" [(ngModel)]="username" />

</div>

<div class="form-group">

<label>Password</label>

<input class="form-control" name="password" [(ngModel)]="password" />

</div>

<div class="form-group">

<label>Message</label>

<input class="form-control" name="message" [(ngModel)]="message" />

</div>

<div class="form-group">

<button class="btn btn-default" (click)="submit()">Отправить</button>

</div>

})

export class AppComponent {

username: string=””;

password: string=””;

message: string=””;

http: HttpClient;

submit(){

const body = {login: username, password: password, message: message};

this.http.post(‘ссылка на API’, body);

}

}

## **Билет 18 +**

### **1. Построение интерфейсов на JSF. Иерархия компонентов JSF.**

● Интерфейс строится из компонентов.

● Компоненты расположены на Facelets-шаблонах или страницах JSP.

● Компоненты реализуют интерфейс javax.faces.component.UIComponent.

● Можно создавать собственные компоненты.

● Компоненты на странице объединены в древовидную структуру — *представление*.

● Корневым элементов представления является экземпляр класса javax.faces.component.UIViewRoot.

Там могут быть как обычные HTML-элементы (div, p, h1, img), так и JSF компоненты. Компоненты — это классы наследники UIComponent. Они образуют иерархию. Корень - UIViewRoot. Существуют сторонние библиотеки компонентов, такие как PrimeFaces и IceFaces, которые упрощают построение интерфейса обширным набором готовых компонентов. Можно создавать свои компоненты.

### **2. Java EE CDI Beans: принципы инъекции бинов.**

В JavaEE главной моделью служит концепция бина, CDI бины поддерживают принцип внедрения зависимостей, чтобы автоматизировать процесс программисту. Для того чтобы объявить бин нужно прописать его в beans.xml, либо использовать аннотации. У бинов есть свои скоупы, которые диктуют их жизненный цикл.

● Универсальные компоненты уровня бизнес-логики.

● Появились в Java EE 6, копируют концепции, реализованные в Spring.

● Общая идея – «отвязаться» от конкретного фреймворка при создании бизнес-логики внутри приложения.

● В большинстве случаев их можно использовать вместо JSF Managed Beans и EJB.

● По реализации очень похожи на JSF Managed Beans.

Для того чтобы внедрить бины можно использовать аннотацию @Inject, тогда контейнер найдет у себя подходящий бин и сам создаст его. Так же если подходит несколько бинов, то будет выброшено исключение, чтобы избежать этого можно использовать аннотацию @Alternative

Внедрение зависимостей работает с полями класса (даже приватными, через рефлексию) и с конструктором (зависимости передаются в конструкторе). Механизм выбора подходящего бина учитывает запрашиваемый класс или интерфейс, название бина (@Named) и альтернативы (@Alternative).

@Qualifier используется когда надо конкретизировать какой именно бин внедрить

В бины внедряется не оригинальный класс бина-зависимости, а класс-прокси, который создается на лету самим контейнером и позволяет реализовывать перехватчики.

### **3. Приложение на Angular, реализующее форму для заполнения бланка на отчисление по собственному желанию. Форма должна принимать на вход имя пользователя и дату, и формировать заполненный бланк заявления (на клиентской стороне)**

@Component({

selector: 'app',

template: `

<form>

<h1>Форма на ПСЖ</h1>

<input type="text" name="firstName" placeholder="Имя" [(ngModel)]="firstName">

<input type="text" name="lastName" placeholder="Фамилия" [(ngModel)]="lastName">

<input type="date" name="lastName" [(ngModel)]="date">

</form>

<main>

<h1>Заявление на отчисление по собственному желанию</h1>

<h2>Ректору ИТМО от {{ firstName }} {{ lastName }}</h2>

Прошу отчислить меня, {{ firstName }} {{ lastName }}, из Университета ИТМО по собственному желанию c {{ date }}.

{{ date }}, подпись: \_\_\_\_\_

</main>

`

})

class AppComponent {

firstName = ""

lastName = ""

date = "01-09-2023"

}

## **Билет 19 +**

### **1. Managed Beans что это, применение, вызов из jsp/xhtml**

Управляемые бины - обычные JAVA классы управляемые JSF. Хранят состояние JSF-приложения. Содержат параметры и методы для обработки данных, получаемых из компонентов. Занимаются обработкой событий.

● Содержат параметры и методы для обработки данных с компонентов.

● Используются для обработки событий UI и валидации данных.

● Жизненным циклом управляет JSF Runtime Envronment.

● Доступ из JSF-страниц осуществляется с помощью элементов EL.

● Конфигурация задаётся в faces-config.xml (JSF 1.X), либо с помощью аннотаций (JSF 2.0).

● Вместо них могут использоваться CDI-бины, EJB или бины Spring.

У Managed Beans есть так называемая “область жизни” (скоуп) – это время, в которое бин будет создан и будет доступен для обработки запросов и выполнения своих задач. Настройка происходит в faces-config.xml или при помощи аннотаций.

Скоупы бывают:

- @NoneScoped - контекст не определён, жизненным циклом управляют другие бины;

- @RequestScoped - контекст - запрос;

- @ViewScoped (JSF 2.0) - контекст-страница(компонент создается один раз при обращении к странице, и используется ровно столько, сколько пользователь находится на странице);

- @SessionScoped - контекст - сессия;

- @ApplicationScoped - контекст - приложение;

- @CustomScoped (JSF 2.0) - компонент создается и сохраняется в коллекции типа Map. Областью жизни управляет программист.

В JSF обращаться к ManagedBean можно через EL:#{myBean.property}

В JSP использовать их не стоит, смешивать два разных фреймворка не стоит.

Сейчас ManagedBean считаются устаревшими, им на смену пришли CDI бины.

### **2. Архитектура и состав Spring MVC**

Фреймворк Spring MVC обеспечивает архитектуру паттерна MVC при помощи слабо связанных готовых компонентов.

1) Model (Модель) инкапсулирует (объединяет) данные приложения, в целом они будут состоять из POJO («Старых добрых Java-объектов», или бинов).● Хранит данные, необходимые для формирования представления. ● Сами по себе эти данные – обычные POJO. ● В общем случае, реализует интерфейс org.springframework.ui.Model.● Есть «упрощённая» реализация,представляющая из себя Map – org.springframework.ui.ModelMap.

2) View (Представление) отвечает за отображение данных фрэймворк не специфицирует жестко технологию на которой будет построено представление. по умолчанию JSP. ● Фреймворк не специфицирует жёстко технологию, на которой должно быть построено представление. ● Вариант «по-умолчанию» – JSP. ● Можно использовать Thymeleaf, FreeMarker, Velocity etc. ● Можно реализовать представление вне контекста Spring – целиком на JS.

3) Controller (Контроллер) обрабатывает запрос пользователя, создаёт соответствующую Модель и передаёт её для отображения на View. ● Класс, который связывает модель с представлением, управляет состоянием модели. ● Помечается аннотацией @Controller. ● Класс или его методы могут быть помечены аннотациями, «привязывающими» их к определённым методам HTTP или URL.

● Spring Web MVC – “базовый” фреймворк в составе Spring для разработки веб-приложений.

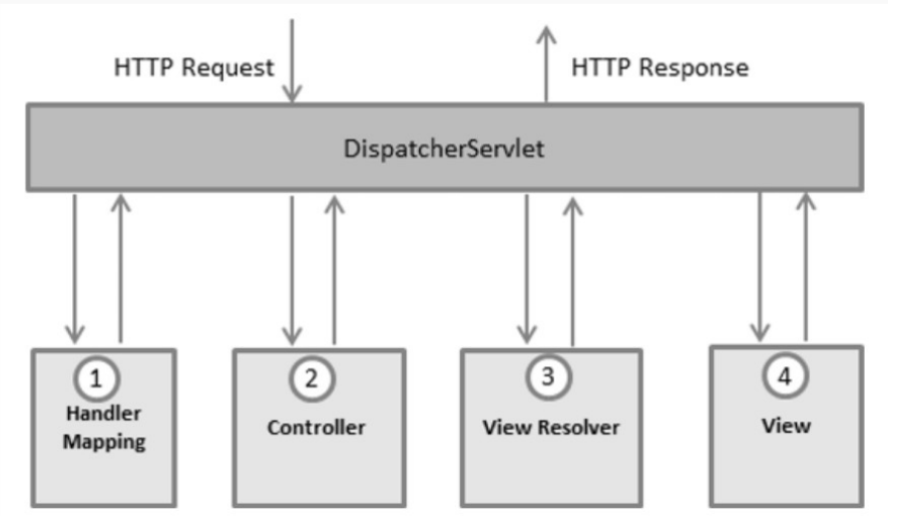
● Основан на паттерне MVC (внезапно!)

● Back-end; универсальный, удобен для разработки REST API.

● На клиентской стороне интегрируется с популярными JS-фреймворками.

● Удобно интегрируется с Thymeleaf.

Архитектура Spring Web MVC

  
Сердце Spring MVC — DispatcherServlet. Это сервлет, который принимает все запросы и передает управление контроллерам, написанными программистом.

Dispatcher Servlet

● Обрабатывает все запросы и формирует ответы на них.

● Связывает между собой все элементы архитектуры Spring MVC.

● Обычный сервлет – конфигурируется в web.xml.  
HandlerMapper — интерфейс для поиска подходящего контроллера

● Механизм, позволяющий распределять запросы по различным обработчикам.

● Помимо «основного» Handler'а, в обработке запроса могут участвовать один или несколько «перехватчиков» (реализаций интерфейса HandlerInterceptor).

● Механизм в общем похож на сервлеты и фильтры.

● «Из коробки» программисту доступно несколько реализаций Handler Mapping.

Контроллер — класс с аннотацией @Controller, который занимается обработкой запросов. В нем реализуется некая бизнес логика для подготовки данных.

● Класс, который связывает модель с представлением, управляет состоянием модели.

● Помечается аннотацией @Controller.

● Класс или его методы могут быть помечены аннотациями, «привязывающими» их к определённым методам HTTP или URL.  
ViewResolver — интерфейс для поиска подходящего представление  
View (представление) — формирует HTML страницу. Чаще всего это шаблон, написанный на JSP, Thymeleaf или Freemaker. Контроллер формирует модель и передает ее представлению. В шаблоне мы можем читать свойства модели и отображать их на странице.

● Представление в Spring Web MVC может быть построено на разных технологиях.

● С каждым представлением сопоставляется его символическое имя.

● Преобразованием символических имён в ссылки на конкретные представления занимается специальный класс, реализующий интерфейс

org.springframework.web.servlet.ViewResolver.

● Существует много реализаций ViewResolver для разных технологий построения представления.

● В одном приложении можно использовать несколько ViewResolver'ов.

### **3. Написать на Angular интерфейс, который проверяет если ли в куки sessionid и если нет, отправляет пользователя на аутентификацию по логину и паролю**

@Component({

selector: 'app',

template: `

<form \*ngIf="!hasCookie">

<h1>авторизация</h1>

<input type="text" name="username" placeholder="Имя">

<input type="password" name="password" placeholder="Пароль">

<input type="submit" value="Войти">

</form>

`

})

class AppComponent {

get hasCookie(): boolean {

return document.cookie.indexOf("jSessionid=") != -1;

}

}

● Реакт интерфейс,который при наличии у юзера куки с id=jSessionid показывает форму для ввода и пароля

import React from 'react';

class App extends React.Component {

constructor(props) {

super(props);

this.state = {

hasCookie: this.checkCookieExists(),

username: '',

password: ''

};

}

checkCookieExists() {

return document.cookie.indexOf('jSessionid=') !== -1;

}

handleInputChange = (event) => {

const { name, value } = event.target;

this.setState({ [name]: value });

}

handleSubmit = (event) => {

event.preventDefault();

const { username, password } = this.state;

// Здесь можно добавить логику для отправки данных на сервер для аутентификации

// Пример: fetch('/auth', {

// method: 'POST',

// body: JSON.stringify({ username, password }),

// headers: {

// 'Content-Type': 'application/json'

// },

// }).then(response => {

// if (response.ok) {

// // Если аутентификация успешна, можно выполнить необходимые действия, например, установить cookie

// document.cookie = 'jSessionid=your\_session\_id\_here';

// this.setState({ hasCookie: true });

// } else {

// // Обработка ошибки аутентификации

// }

// }).catch(error => {

// console.error('Error:', error);

// });

document.cookie = 'jSessionid=your\_session\_id\_here';

this.setState({ hasCookie: true });

}

render() {

const { hasCookie, username, password } = this.state;

if (hasCookie) {

return <div>Пользователь аутентифицирован</div>;

}

return (

<form onSubmit={this.handleSubmit}>

<h1>Авторизация</h1>

<input

type="text"

name="username"

placeholder="Имя"

value={username}

onChange={this.handleInputChange}

/>

<input

type="password"

name="password"

placeholder="Пароль"

value={password}

onChange={this.handleInputChange}

/>

<input type="submit" value="Войти" />

</form>

);

}

}

export default App;

## **Билет 20 +**

### **1. Контекст управляемых бинов. Конфигурация контекста бина**

Контекст определяет, к чему будет привязан бин и его время жизни: ко всему приложению, сессии, запросу, JSF-странице.

Конфигурировать можно через аннотации, либо через faces-config.xml: <managed-bean-scope>application</managed-bean-scope>

- @NoneScoped - контекст не определён, жизненным циклом управляют другие бины;

- @RequestScoped - контекст - запрос;

- @ViewScoped (JSF 2.0) - контекст-страница(компонент создается один раз при обращении к странице, и используется ровно столько, сколько пользователь находится на странице);

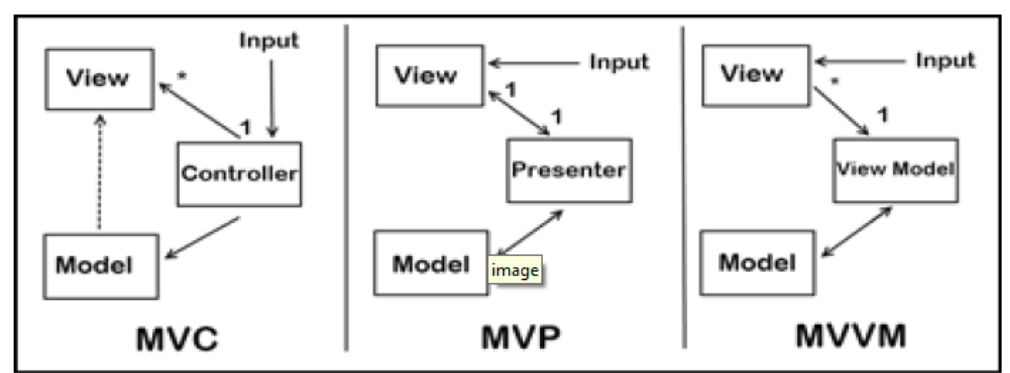
- @SessionScoped - контекст - сессия;

- @ApplicationScoped - контекст - приложение;

- @CustomScoped (JSF 2.0) - компонент создается и сохраняется в коллекции типа Map. Областью жизни управляет программист.

Дополнительно по управляемым бинам:  
У ManagedBean могут иметь ManagedProperty (управляемые свойства), в которые фреймворк может внедрять определенные начальные значения (строковые, числовые, enum). Конфигурируется это через аннотации ManagedProperty или в файле faces-config.xml:  
<managed-bean>  
 <managed-bean-name>myBean</managed-bean-name>  
 <managed-bean-class>my.package.MyBean</managed-bean-class>  
 <managed-bean-scope>application</managed-bean-scope>  
 <managed-property>  
 <property-name>age</property-name>  
 <value>27</value>  
 </managed-property>  
</managed-bean>  
Удобство в том, что эти свойства можно переопределять в контейнере, без необходимости пересборки классов и редеплоинга приложения.  
Сами бины при этом управляются фреймворком JSF. У него есть свой контекст FacesContext, при помощи которого (если очень хочется) можно программным путем вытащить нужный бин

### **2. Шаблоны MVVM и MVP. Сходства и отличия от MVC**



MVC.

Механизм: Инпут на контроллер, контроллер взаимодействует с моделью. Модель обновляет вью.

MVP, или Model-View-Presenter - шаблон, созданный много позже MVC. Вместо Контроллера- Презентер. отвечает за отрисовку и обновление View, а за обновление Model. содержит логику интерфейса и отвечает за синхронизацию Model и View.

Механизм: Инпут от View, презентер реагирует на ивенты от View и обновляет модель. Далее презентер реагирует на ивенты изменения состояния и обновляет View.

MVVM - Model-View-ViewModel - ещё один шаблон. Смысл в том, что ViewModel не связан напрямую с View, а общается с ним с помощью простых команд, и вью подписывается на его изменение. Сама же ViewModel содержит модель, преобразованную к представлению, а также команды, через которые представление обращается к модели.

Механизм:

У вьюмодел нет ссылки на вью, а вью подписывается на изменения вьюмодел(синхронизируется автоматически через data binding). Вьюмодел же берет данные из модели и уведомляет ее об изменениях

Таблица:

MVC

- Входные данные направлены на контроллер

- Отношение "Многие к одному" между контроллером и представлением

- Представление не имеет никаких сведений о контроллере

- Контроллер передает модель представлению, поэтому представление имеет сведения о контроллере

MVP

- Ввод начинается с представления

- Взаимно однозначное сопоставление между представлением и связанным с ним презентатором

- Представление содержит ссылку на презентатора; Презентатор также реагирует на события, запускаемые из представления, поэтому презентатор осведомлен о представлении

- Представление не осведомлено о модели

MVVM

- Ввод начинается с model

- Сопоставление "Один ко многим" между различными и одной ViewModel

- Представление содержит ссылку на ViewModel; Модель представления не содержит информации о представлении.Здесь разные технологии могут совместно использовать ViewModel, т.е. WCF и Silverlight могут совместно использовать одну и ту же ViewModel.

- View не поддерживает модель

### **3. Компонент для React, формирующий строку с автодополнением. Массив значений для автодополнения должен получаться с сервера посредством запроса к REST API**

function Autocomplete() {

const [candidates, setCandidates] = useState([])

async function updateList(e) {

let res = await fetch("http://api.itmo.ru/students/search?q=" + e.target.value);

setCandidates(await res.json())

}

return <div>

<input type="text" onChange={updateList}>

<ul>

{ candidates.map((text, i) => <li key={i}>{text}</li>) }

</ul>

</div>;

}

## **Билет 21 +**

### **1. REST в спринге: методы и аргументы**

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 = @Controller + @ResponseBody

REST можно реализовать в обычном Spring MVC контроллере, используя аннотацию @Controller. На каждый метод-обработчик, возвращающий сериализованный ответ в теле, нужно добавить аннотацию @ResponseBody. Для удобства в Spring сделали аннотацию @RestController, которая по умолчанию применяет @ResponseBody к всем методам, помеченными аннотацией @RequestMapping.

@RestController — говорит спрингу, что данный класс является REST контроллером. Т.е. в данном классе будет реализована логика обработки клиентских запросов.

/\*

для обработки запросов нужно написать класс со спец. аннотацией @RestController  
в котором будет реализована логика обработки клиентских запросов.

специальными аннотациями @Get/Post/Put/DeleteMapping помечаются методы для обработки http запросов. В аргументах аннотации можно указать path, по которому можно обратиться к данному методу. @RequestBody помечается десериализованный из запроса объект. DTO(data transfer object).

\*/

Методы-обработчики запросов могут принимать параметры из URL через аннотацию @PathVariable, параметры GET-запроса (query string) через @RequestParam, десериализованные данные из тела запроса через @RequestBody. Spring автоматически определит формат данных, исходя из заголовка Content-Type и десериализует их, при наличии соответствующей библиотеки (для JSON используется Jackson)

@RequestBody, - десереализует автоматически жававский объект из JSON

@ResponseBody - сереализует в JSON для передачи клиенту, указывает, что тип результата должен быть записан прямо в теле ответа в любом указанном вами формате, например JSON или XML.

@RequestParam можно использовать для обработки переменных шаблона в сопоставлении URI запроса и задавать их в качестве параметров метода, можно вытащить часть GET параметра

@PathVariable используется для привязки параметра метода к переменной path, можно использовать для извлечения динамической части URL, когда данные приходят как часть URL

### **2. Навигация в React. React Router**

React Router - библиотека для декларативной маршрутизации в React приложениях.

React Router - система маршрутизации, позволяющая делать навигацию между компонентами, а также позволяет сопоставлять запросы к с определенными компонентами. Тк React - SPA framework то браузер всегда показывает одну и ту же страницу. Содержимое страницы меняется динамически при нажатии пользователя на ссылку.

Три основных объекта:

- Router определяет набор маршрутов и выполняет сопоставление запроса с маршрутами. Выбирает маршрут для обработки запроса по URL

- Routes содержит набор маршрутов и позволяет выбрать первый попавшийся маршрут по нужному URL и его использовать для обработки.

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

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

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

- <Link> вместо <a> (<Link to='/admin'>Admin</Link>)

- Если делаете навигацию, то используйте <NavLink>.

- Переход без Link (программный) (his.props.history.push(`/user/${value}`))

### **3. Реализовать бронь авиабилетов на jsf**

@ManagedBean

@SessionScoped

@Setter

@Getter

public class Ticket {

private int id;

private int price;

private String source;

private String dest;

@ManagedProperty(value = "#{repoBean}")

private TicketRepo repo;

private List<Ticket> available;

@PostConstruct void init(){  
available = repo.getAvaible();

}

public void book(int id) {

repo.book(id);

available = repo.getAvailable()

}

}

<!DOCTYPE html>

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

xmlns:ui="http://java.sun.com/jsf/facelets"

xmlns:h="http://java.sun.com/jsf/html">

<head>

<title>Бронь авиабилетов</title>

</head>

<body>

<h1>Бронь авиабилетов> </h1>

<ui:repeat value="#{tickets.available}" var="ticket">

<div class="ticket">

#{ticket.source} - #{ticket.destination}

Номер билета: #{ticket.id}

Цена: #{ticket.price} руб

<h:commandButton action="#{tickets.book(ticket.id)}" value="Забронировать"/>

</div>

</ui:repeat>

</body>

</html>

## **Билет 22**

### **1. Класс FacesServlet - назначение, особенности конфигурации**

FacesServlet — главный сервлет, который занимается жизненным циклом обработки запросов в веб приложениях, построенных на JSF фреймворке.

-Обрабатывает запросы с браузера

-Формирует объекты события и вызывает методы-слушатели.

Является частью фреймворка, самому его реализовывать не нужно.  
Конфигурация FacesServlet производится через файл web.xml.

<!-- Faces Servlet -->

<servlet>

<servlet-name>Faces Servlet</servlet-name>

<servlet-class>

javax.faces.webapp.FacesServlet

</servlet-class>

<load-on-startup>1</load-on-startup>

</servlet>

<!-- Faces Servlet Mapping -->

<servlet-mapping>

<servlet-name>Faces Servlet</servlet-name>

<url-pattern>/faces/\*</url-pattern>

</servlet-mapping>  
В нем настраиваются правила навигации, регистрируются ManagedBeans, конвертеры, валидаторы, свои компоненты пользовательского интерфейса. Многие из этих настроек также доступны через аннотации.

В разделе <servlet> регистрируется сам сервлет и ему выдается имя, а в разделе <servlet-mapping> правила url навигации

### **3. REST - контроллер на Spring Web MVC, предоставляющий CRUD-интерфейс к таблице со списком покемонов**

@RestController

class PokemonResource {

@Autowired

private PokemonRepository repository;

@GetMapping("/pokemons")

List<Pokemon> all() {

return repository.findAll()

}

@GetMapping("/pokemons/{id}")

Pokemon one(@PathVariable Long id) {

return repository.findOne(id)

}

@PostMapping("/pokemons")

Long createNew(@RequestBody Pokemon pokemon) {

return repository.save(pokemon)

}

@DeleteMapping("/pokemons/{id}")

void delete(@PathVariable Long id) {

repository.delete(id)

}

}

## **Билет 23 +**

### **1. Конвертеры JSF, создание и назначение**

Конверторы в JSF — классы, реализующие интерфейс javax.faces.convert.Converter. Используются для преобразования данных компонента в заданный формат (дата, число и т. д.). Методы: getAsObject и getAsString. Существуют встроенные конвертеры для основных типов. (IntegerConverter, DoubleConverter, DateTimeConverter, EnumConverter)

Чтобы зарегистрировать свой конвертер, необходимо прописать его в faces-config.xml либо воспользоваться аннотацией FacesConverter

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>  
Назначение в JSF:

-автоматическое(на основе типа данных)

-с помощью атрибута converter <h:inputText converter="#{javax.faces.DateTime}"/>

-с помощью вложенного тега <h:outputText value="#{user.birthDay}"> <f:converter

converterId="#{javax.faces.DateTime}"/> </h:outputText>

### **2. Реализация model в Spring web MVC**

Model в Spring MVC — интерфейс, для работы с данными необходимыми для формирования представления. Основные методы модели — addAttribute, getAttribute, asMap. В модель можно класть любой объект(POJO), и доставать его по строковому ключу. Через модель представление получает доступ к данным приложения, которые необходимо вывести на странице.

### Model (Модель) инкапсулирует (объединяет) данные приложения, в целом они будут состоять из POJO («Старых добрых Java-объектов», или бинов).● Хранит данные, необходимые для формирования представления. ● Сами по себе эти данные – обычные POJO. ● В общем случае, реализует интерфейс org.springframework.ui.Model.● Есть «упрощённая» реализация,представляющая из себя Map – org.springframework.ui.ModelMap.

Существует аннотация @ModelAttribute, которая позволяет задать в контроллере метод, заполняющий модель атрибутами, которые потом будут доступны из всех обработчиков. Также можно пометить аргумент обработчика как @ModelAttribute. В таком случае, наш объект будет создан, используя параметры из запроса, а затем будет положен в модель.

Чтобы создать бин, мы должны использовать либо аннотации (@Component для класса, @Bean для метода класса @Configuration), либо старый добрый XML (<bean class=”...” name=”...”/>). Чтобы “подключить” бин к контроллеру, можно воспользоваться аннотацией @Autowired. Контроллер сам вытащит нужный бин из контейнера. Можно внедрить бин через конструктор контроллера, через метод-сеттер, и, конечно, через старый добрый XML.

## **Билет 24 +**

### **1. CDI Beans**

● Универсальные компоненты уровня бизнес-логики.

● Появились в Java EE 6, копируют концепции, реализованные в Spring.

● Общая идея – «отвязаться» от конкретного фреймворка при создании бизнес-логики внутри приложения.

● В большинстве случаев их можно использовать вместо JSF Managed Beans и EJB.

● По реализации очень похожи на JSF Managed Beans.

CDI bean — класс, удовлетворяющим следующим требованиям: класс должен быть статическим (а не вложенным не статическим) и должен иметь конструктор без аргументов  
CDI beans появились начиная с Java EE 7 и поддерживают гораздо больше фич, чем JSF ManagedBeans(перехватчики, стереотипы, декораторы и многое другое). CDI бины, как и их аналоги, поддерживают внедрение зависимостей. Жизненным циклом бинов управляет CDI контейнер, то есть их экземпляры создаются автоматически при необходимости. CDI beans максимально абстрактная реализация паттерна CDI и совместимы со всеми фрэймворками

### **2. Angular DI**

Dependency Injection - паттерн проектирования (сокращенно DI), который позволяет создавать объект, использующий другие объекты. При этом поля объекта настраиваются внешней сущностью.

• Компоненты могут использовать сервисы с помощью DI.

• Для того, чтобы класс можно было использовать с помощью DI, он должен содержать декоратор @Injectable().

Основные принципы реализации DI:

• Приложение содержит как минимум один глобальный Injector, который занимается DI.

• Injector создаёт зависимости и передаёт их экземпляры контейнеру (container).

• Провайдер (provider) -- это объект, который сообщает Injector’у, как получить или создать экземпляр зависимости.

• Обычно провайдером сервиса является сам его класс.

• Зависимости компонентов указываются в качестве параметров их конструкторов

Самая главная фича Angular что он поддерживает DI.

Чаще всего в компоненты внедряют сервисы, в которых реализуется бизнес логика, не связанная с представлением. Например логирование, общение с API. (с помощью DI легче тестировать). Можно писать свои классы помеченные аннотацией @Injectable и внедрять их. Зависимости передаются ввиде параметров конструктора.

### **3. JSF страничка с данными из бина**

@Named(“myBean”)

@ManagedBean  
class MyBean {

int value;

public int getValue() {  
 return value;  
 }  
}  
  
<h:outputText value=”#{myBean.value}”/>

## **Билет 25 +**

### **1. Шаблоны и представление в Angular**

Компоненты + шаблоны = представление

-Представление view компонента задается с помощью шаблона (template)

-Представления группируются иерархически.

-Компонент может содержать иерархию представлений (view hierarchy) которая содержит встроенные представления (embedded views) из других компонентов.

● Похож на обычный HTML.

● Взаимодействие с классом компонента осуществляется с помощью ссылок на его свойства *(data binding)*.

● Также можно использовать *фильтры (pipes)* и *директивы (directives)*.

Связь между шаблоном и свойствами компонента -- двунаправленная.

Три вида связей:

● {{hero.name}} -- отображение (interpolation). Показывает значение свойства в HTML-разметке.

● [hero] -- связывание свойства (property binding). Передаёт значение свойства selectedHero родительского компонента HeroListComponent в качестве свойства hero дочернего компонента HeroDetailComponent.

● (click) -- связывание по событию (event binding). Вызывает метод, когда пользователь кликает по элементу.

Фильтры (pipes)

● Позволяют осуществлять преобразование формата отображаемых данных (например, дат или денежных сумм) прямо в шаблоне.

● Фильтры можно объединять в последовательности (pipe chains).

● Фильтры могут принимать аргументы.

Директивы (directives)

● Инструкции по преобразованию DOM.

● Создаются с помощью декоратора @Directive().

● Все компоненты, технически -- директивы.

● Два вида:

○ Структурные директивы (structural directives):

<li \*ngFor="let hero of heroes"></li>

<app-hero-detail \*ngIf="selectedHero"></app-hero-detail>

○ Директивы-атрибуты (attribute directives):

<input [(ng Model)]="hero.name">

-Представление компонента(как будет отображаться) задается с помощью шаблонов.

Шаблоны похожи на обычный хтмл, взаимодействуют с классом компонента через data binding.

-Представления группируются иерархически.

-Компонент может содержать иерархию представлений (view hierarchy) которая содержит встроенные представления (embedded views) из других компонентов.

Поддерживается интерполяция: {{ value }}. Синтаксис атрибутов расширен: [attr]=”value” — одностороннее связывание, [(attr)]=”value” — двухстороннее связывание, @event=”handler” — обработчик событий. Для условной отрисовки используется директива \*ngIf, для циклов — \*ngFor.

Шаблоны могут содержать фильтры и директивы.

Директивы - инструкции по преобразованию DOM

Фильтры- могут преобразовывать данные в нужный формат. Поддерживаются pipe chains, также могут принимать аргументы

### **2. Dependency Lookup Spring**

Dependency Lookup (DL) — подход при котором компонент напрямую просит у контейнера передать ему зависимость. Противопоставляется DI, который происходит автоматически. Не рекомендуется к использованию, но необходимо когда DI дает сбой (например достает не тот бин).  
В Spring для реализации DL нужно сначала получить контекст приложения (например ClassPathXmlApplicationContext). Затем: context.getBean(“name”) — вернет нужный бин. Также можно указать конкретный класс или интерфейс: context.getBean(“name”, MyBean.class)

Пример:

ApplicationContext appContext = new ClassPathXmlApplicationContext(“/application-context.xml”);

MyBean bean = appContext.getBean(“myBean”);

То есть, мы сами достаём xml и сами грузим оттуда нужный бин. Но так почти никогда не делают.

### **3. Конфигурация, чтобы JSF обрабатывал все запросы приходящие с .xhtml и со всех URL, начинающихся с /faces/**

web.xml

<?xml version="1.0" encoding="UTF-8"?>

<web-app xmlns="http://xmlns.jcp.org/xml/ns/javaee"

xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"

xsi:schemaLocation="http://xmlns.jcp.org/xml/ns/javaee http://xmlns.jcp.org/xml/ns/javaee/web-app\_4\_0.xsd"

version="4.0">

<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 26 +**

### **1. Process validation phase, Update module values phase**

Process Validation Phase

● Вызываются валидаторы, зарегистрированные для компонентов представления.

● Если значение компонента не проходит валидацию, формируется сообщение об ошибке, которое сохраняется в FacesContext.

В фазе Process Validations вызывается метод UIViewRoot#processValidators(), которые для каждого компонента дерева вызывает метод UIComponent#processValidators(). Компоненты с установленным в true свойством immediate уже выполнили свою валидацию на предыдущем шаге.

Если на данном **этапе жизненного цикла** возникнут ошибки преобразования или валидации, то будет вызван метод FacesContext#addMessage для добавления сообщения об ошибке соответствующих компонентов, и их свойство valid будет выставлено в false.

Если в данной фазе какой-либо из вызванных методов validate() или слушатель событий (event listener) вызвал один из методов

- FacesContext#responseComplete(), то обработка текущего запроса должна быть немедленно прекращена

- FacesContext#renderResponse(), то управление должно немедленно перейти к фазе Render Response.

Иначе управление должно перейти к фазе Update Model Values.

Update Module Values Phase

● Если данные валидны, то значение компонента обновляется.

● Новое значение присваивается полю объекта компонента.

На данном этапе можно константировать, что параметры запроса корректны, и что локальное значение каждого компонента дерева с корневым элементом UIViewRoot было обновлено. Наступило время обновления данных в модели приложения для подготовки к выполнению событий приложения, которые находятся в очереди.

В данной фазе вызывается метод UIViewRoot#processUpdates(), после чего для каждого компонента дерева будут вызваны методы UIComponent#processUpdates(). Обновление модели выполняется в методе updateModel() компонента. В процессе обновления модели в очередь могут быть добавлены события, которые будут будут распространены заинтересованным слушателям событий.

На завершающей стадии данного этапа все объекты модели будут обновлены значениями, содержащимися в соответствующих компонентах.

Если какой-либо из вызванных методов validate() или слушатель событий (event listener) на данном этапе вызвал один из методов

- FacesContext#responseComplete(), то обработка текущего запроса должна быть немедленно прекращена

- FacesContext#renderResponse(), то управление должно немедленно перейти к фазе Render Response.

Иначе управление должно перейти к фазе **Invoke Application**.

### **2. Жизненный цикл Spring-приложения**

1. Создается контейнер  
2. Контейнер считывает конфиг  
3. Загружается описание бинов из конфига и classpath (с аннотацией)  
4. Создаются экземпляры бинов  
5. Производится внедрение зависимостей  
6. Вызывается PostConstructor  
7. Вызываются callback’и BeanPostProcessor. обновление

8. Приложение запущено  
9. Приложение закрывается  
10. Вызываются PreDestroy, callback’и  
11. Приложение остановлено

Подробно:

Жизненный цикл сприга до создания объекта:

1)Парсинг конфигурации, создание beandefinition Конфигурация, написанная на xml или в виде Java кода …, попадает в BeanDefinitionReader, объект, который парсит конфигурация в том или ином виде. После считывания он выдает BeanDefinitionы. Сам BeanDefinition хранит всю информацию о том, каким образом спрингу создавать бины, как бин определен, какие зависимости в себе содержит, какие аннотацию и другие настройки.

2) настройка beandefinition

Если Bean настроен как Singleton (этот тип бинов у спринга по умолчанию), то он в конвейере будет обработан сразу, т.е его прогонят по жизненному циклу бина, будет создан объект и он попадет в IoC контейнер.

2.1) Если Singleton но Lazy Bean - не будет создан сразу, только при необходимости (по запросу). - он будет создан и помещен в контейнер.

2.2) Prototype (не Singleton) - создается по запросу. Они не попадают в IoC контейнер (так сказал Письмак). Спринг управляет им до момента как мы его заполучили. Как только он создан и отдан, он перестает управлять его жизненным циклом. ~ Т.е выполнение инструкций при при удалении будет недоступно.

3)создание кастомных FactoryBean

4)создание экземпляров бинов. BeanFactory создает экземпляры бинов делегируя создание FactoryBean если мы его определили

5)Настройка созданных бинов. Устанавливаются системные переменные, property для того, чтобы в дальнейшем правильно создать бины.(параметры по умолчанию)

Жизненный цикл бина:

1. Инстанцирование объекта. Техническое начало жизни бина, работа конструктора его класса; - описано выше

2. Установка свойств из конфигурации бина, внедрение зависимостей; AutoWired и т.д

3.~ Нотификация aware-интерфейсов. BeanNameAware, BeanFactoryAware и другие.

Обновление контекста: вызов постпроцессоров (когда бин уже создан его допиливаем):

4. Пре-инициализация – метод postProcessBeforeInitialization() интерфейса BeanPostProcessor;

5. Инициализация. Разные способы применяются в таком порядке:

• Метод бина с аннотацией @PostConstruct из стандарта (рекомендуемый способ);

• Метод afterPropertiesSet() бина под интерфейсом InitializingBean;

• Init-метод. Для отдельного бина его имя устанавливается в параметре определения initMethod. В xml-конфигурации можно установить для всех бинов сразу, с помощью default-init-method;

6. Пост-инициализация – метод postProcessAfterInitialization() интерфейса BeanPostProcessor.

Уничтожение:

1. Метод с аннотацией @PreDestroy;

2. Метод с именем, которое указано в свойстве destroyMethod определния бина (или в глобальном default-destroy-method);

3. Метод destroy() интерфейса DisposableBean.

### **3. Интерфейс на Angular, который выводит интерактивные часы с обновление каждую секунду**

import { Component } from '@angular/core';

@Component({

selector: 'my-app',

template: <div>{{ time }}</div>

})

export class AppComponent {

time: ""

setInterval(() => this.time = new Date(), 1000)

}

## **Билет 27 +**

### **1. Реализация Ajax в jsf**

в JSf есть два способа реализации Ajax запросов: старый и новый.

1 способ: JavaScript API - **jsf.ajax.request()** (детальный контроль)

- event - событие, по которому отправляется AJAX-запрос

- execute - компоненты, обрабатываемые в цикле обработки запроса

- render - компоненты, подлежащие перерисовке

<h:commandButton id="submit" value="submit"

onclick="jsf.ajax.request(this, event, {execute: 'myInput', render: 'outText'});">

</h:commandButton>

2 способ: тег **<f:ajax>** (декларативное добавление AJAX)

- execute - компоненты, обрабатываемые в цикле обработки запроса

- render - компоненты, подлежащие перерисовке

<h:commandButton id="submit" value="submit">

<f:ajax execute="@form" render="msg"/>

</h:commandButton>

### **2. CDI beans: контекст (Bean Scope)**

Аналог контекста (scope) управляемых бинов JSF.

Определяет жизненный цикл бинов и их видимость друг для друга.

- @RequestScoped - контекст - запрос;

- @ViewScoped - контекст-страница(компонент создается один раз при обращении к странице, и используется ровно столько, сколько пользователь находится на странице); из JSF, но тоже работает

- @SessionScoped - контекст - сессия;

- @ApplicationScoped - контекст - приложение;

- @ConversationScoped - Областью жизни управляет программист. Управление осуществляется через инъекцию объекта javax.enterprise.context.Conversation.

- @Dependent. Используется по умолчанию. ЖЦ определяется тем где он был использован

### **3. Интерфейс реализации логин+пароль на React. На стороне сервера- Rest API**

function App(props) {

const [login, setLogin] = useState("")

const [password, setPassword] = useState("")

async function onSubmit() {

let res = await fetch("api/login", {

method: "POST",

headers: {"Content-Type": "application/json"},

body: JSON.stringify({login: login, password: password})

})

let token = await res.text();

localStorage.setItem("token", token)

}

return (

<form onSubmit={onSubmit}>

<input type="text" placeholder="Логин" required value={login}

onChange={e => setLogin(e.target.value)}/>

<input type="password" placeholder="Пароль" required value={password}

onChange={e => setPassword(e.target.value)}/>

<input type="submit" value="Войти"/>

</form>

);

}

## **Билет 28 +**

### **1. Фаза получения значений компонентов (Apply Request Values Phase)**

● На стороне клиента все значения хранятся в строковом формате — нужна проверка их корректности: Вызывается конвертер в соответствии с типом данных значения.

● Если конвертация заканчивается успешно, значение сохраняется в локальной переменной компонента.

● Если конвертация заканчивается неудачно, создаётся сообщение об ошибке, которое помещается в FacesContext.

На данном **этапе жизненного цикла** каждый компонент обновляет своё состояние на основании информации текущего запроса (параметры, заголовки, cookies и т.д.). Фреймворк должен вызвать метод UIViewRoot#processDecodes, который вызовет методы UIComponent#processDecodes для всех компонентов дерева. Для UIInput-компонентов будет выполнено преобразование данных.

В процессе декодирования данных запроса компоненты могут вызвать определенные действия :

- компоненты, реализующие интерфейс ActionSource (например, UICommand), выполняют проверку своей активности и, при необходимости, добавят ActionEvent в очередь событий. Данное событие будет вызвано в конце фазы Apply Request Values или в конце фазы Invoke Application, в зависимости от состояния свойства immediate у активированного компонента.

- компоненты, реализующие интерфейс EditableValueHolder (например, UIInput) с установленным свойством immediate в true, вызовут преобразование и валидацию (включая потенциально запуск событий ValueChangeEvent, которые по-нормальному происходят в фазе Process Validations).

В конце этой фазы все компоненты, которые реализуют интерфейс EditableValueHolder, будут обновлены в соответствии с данными, переданными в запросе. Если при преобразовании и валидации возникнут ошибки, то с помощью метода FacesContext#addMessage для соответствующих компонентов будут добавлены сообщения, и их свойство valid будет выставлено в false.

Если в данной **фазе жизненного цикла** какой-либо из слушателей событий (event listener) или методов decode() вызвал один из методов

- FacesContext#responseComplete(), то обработка текущего запроса должна быть немедленно прекращена

- FacesContext#renderResponse(), то управление должно немедленно перейти к фазе Render Response.

Иначе управление должно перейти к фазе Process Validations.

### **2. Реализация IoC и CDI в Spring**

Инверсия управления — принцип используемый для уменьшения связности кода. Заключается в том, что самописным кодом управляет общий фреймворк, занимающийся жизненным циклом компонентов и коммуникацией между ними. Чаще всего реализуется посредством внедрения зависимостей.

Внедрение зависимостей - паттерн проектирования (сокращенно DI), который позволяет создавать объект, использующий другие объекты. При этом поля объекта настраиваются внешней сущностью.   
Внедрение зависимостей — позволяет убрать зависимость компонента от контейнера.

● Не требуется реализации каких-либо интерфейсов.

● Не нужны прямые вызовы API.

В спринге ApplicationContext представляет контейнер который управляет жизненным циклом компонентов и коммуникацией между ними.

Внедрение зависимостей — позволяет убрать зависимость компонента от контейнера. компоненты ничего не знают о своих зависимостях и пользуются лишь интерфейсами. Конкретные реализации передаются через конструктор, поля, или сеттеры. Реализуется через аннотации.

За внедрение зависимостей отвечает аннотация @Autowired.

DI:

● когда нужен бин — он запрашивается у ApplicationContext;

● до Spring Framework 4.3 нужно было явно указывать аннотацию @Autowired;

● «по умолчанию» — внедрение через конструктор;

● используя @Autowired — можно внедрять через setter;

● также поддерживаются аннотации из JSR-330 — @Inject, @Named.

Spring IoC контейнеру требуются метаданные для конфигурации. Для этого классы помечаются аннотацией @Component, а также её наследниками @Repository, @Service и @Controller

## **Билет 29 +**

### **1. Spring MVC: handler mapping**

Вся логика работы Spring MVC построена вокруг DispatcherServlet, который принимает и обрабатывает все HTTP-запросы и ответы на них. При получении запроса, происходит следующая цепочка событий:

1. DispatcherServlet обращается к интерфейсу HandlerMapping, который определяет какой контроллер должен быть вызван ()

2. Контроллер принимает запрос и вызывает соответствующий служебный метод (GET, POST), который возвращает в диспатчер имя View

3. При помощи ViewResolver диспатчер определяет, какой View надо использовать на основании полученного имени

4. После того, как View создан, диспатчер отправляет данные модели в виде атрибутов, которые уже в конечном итоге отображаются в браузере

Когда DispatcherServlet получает запрос, он на основании конфигурации HandlerMapping выбрать на какой контроллер пойдет запрос.

● Механизм, позволяющий распределять запросы

по различным обработчикам.

● Помимо «основного» Handler'а, в обработке запроса могут участвовать дополнительные

«перехватчики» (реализаций интерфейса HandlerInterceptor).

● Механизм в общем похож на сервлеты и фильтры.

● «Из коробки» программисту доступно несколько реализаций Handler Mapping

### **2. JSX. Применение в реакте. Пример синтаксиса**

Разметка страницы в React-приложениях обычно выполняется с использованием JSX (JavaScript XML), который предоставляет удобный способ описания пользовательского интерфейса. JSX похож на HTML, но использует синтаксис JavaScript для описания компонентов.

Точкой входа (корнем) являет index.js который определяет компоненты. Каждый компонент включает в себя другие компоненты.

В React приложениях разметка пишется в JSX файлах. JSX — надстройка над JS, которая позволяет вкраплять HTML-синтаксис в код. Можно использовать стандартные HTML элементы (такие как div, span, h1, input) так и кастомные React компоненты.  
JSX код: <div className=”foo”>text</div> компилируется в вызов функции React.createElement(“div”, { className: “foo”}, “text”)

JSX:

● Расширение языка JavaScript

● Cахар для React.createElement(component, props, ...children)

● Компилируется Babel’ом в JS

● визуально близок к HTML

Правила JSX

Возвращает один корневой элемент

Чтобы вернуть несколько элементов из компонента, оберните их одним родительским тегом (например <div> или вместо них можно написать <> и </> (называется фрагмент))

JSX требует, чтобы теги были явно закрыты: самозакрывающиеся теги, такие как <img> должны стать <img />, а оберточные теги типа <li>oranges должны быть записаны как <li>oranges</li>. В React многие атрибуты записываются в camelCase.

- JSX — переиспользование компонентов

class NameForm extends React.Component {

render() {

return (

<div> smth </div>

);

}}

class ComponentThatUseAnotherComponent extends

React.Component {

render() {

return (

<div>

We use component from previous slide

<NameForm/>

</div>

);

}}

Важно: Пользовательские компоненты должны быть написаны с большой буквы

В фигурных скобках ‘{}’ JS выражения. Т.к. это выражения никаких if, for, объявлений и прочего.

render() {

let name = ‘Vasya’;

return (

<div>

<h3>Name { this.name}</h3>

<TodoList items={ this.state.items} />

</div>

);

}

Условный рендер в JSX

class Greeting extends React.Component {

render() {

const isLoggedIn = props.isLoggedIn;

if (isLoggedIn) { return <UserGreeting />; }

return <GuestGreeting />;

}}

Циклы в JSX

function Item(props) { return <li>{props.message}</li>; }

function TodoList() {

const todos = ['finish doc', 'submit pr'];

return (

<ul>

{todos.map((message) => <Item key={message}

message={message} />

)}

</ul>

);

}

(Jsx потом компилируется в JS с помощью Babel)

import React from 'react';

import CustomButton from './CustomButton';

function WarningButton() {

// return React.createElement(CustomButton, {color: 'red'}, null);

return <CustomButton color="red" />;

}

### **3. Написать на JSF правило навигации в faces-config.xml со страницы page1.jsp на page2.jsp с использованием этой кнопки: <h:commandButton action="goToPage" value="Go to Page 2"/>**

**faces-config.xml**

<?xml version="1.0"?>

<!DOCTYPE faces-config PUBLIC "-//Sun Microsystems, Inc.//DTD JavaServer Faces Config 2.0//EN"

"http://xmlns.jcp.org/xml/ns/javaee/web-facesconfig\_2\_3.xsd">

<faces-config version="2.3" xmlns="http://xmlns.jcp.org/xml/ns/javaee">

<navigation-rule>

<from-view-id>/page1.jsp</from-view-id>

<navigation-case>

<from-outcome>goToPage</from-outcome>

<to-view-id>/page2.jspl</to-view-id>

<redirect/>

</navigation-case>

</navigation-rule>

</faces-config>

## **Билет 30 +**

### **1. Компоненты в Angular: взаимодействие с представлениями и сервисами**

В модулях определяются компоненты. Компоненты — строительные блоки интерфейса, кирпичики, инкапсулирующие верстку, стили, и логику приложения, которые можно переиспользовать внутри других компонентов.

**Компонент**

**Компонент** - обособленная часть функционала со своей логикой, HTML-шаблоном и CSS-стилями. Компонент управляет отображение представление на экране.

За объявление компонента отвечает декоратор @Component()

Основные свойства объекта, который принимает декоратор:

•selector — название компонента;

•template (или templateUrl) — HTML-разметка в виде строки (или путь к HTML-файлу);

•providers — список сервисов, поставляющих данные для компонента;

•styles — массив путей к CSS-файлам, содержащим стили для создаваемого компонента.

ЖЦ компонента:

**OnChanges** - устанавливаются или изменяются значения входных свойств класса компонента;

**OnInit** - устанавливаются "обычные" свойства; вызывается единожды вслед за первым вызовом

OnChanges();

**DoCheck** - происходит изменения свойства или вызывается какое-либо событие;

**AfterContentInit** - в шаблон включается контент, заключенный между тегами компонента;

**AfterContentChecked** - аналогичен DoCheck(), только используется для контента, заключенного между тегами компонента;

**AfterViewInit** - инициализируются компоненты, которые входят в шаблон текущего компонента;

**AfterViewChecked** - аналогичен DoCheck(), только используется для дочерних компонентов;

**OnDestroy** - компонент "умирает", т.е. удаляется из DOM-дерева

Любой компонент состоит из TS класса, помеченного декоратором @Component, HTML шаблона, CSS стилей. Компонент вместе с шаблоном образуют представление. Представления образуют иерархию. Существует двухсторонняя связь между классом компонента и представлением(data binding) — при изменении данных в компоненте обновляется представление и наоборот.  
Задачи приложения, которые не касаются представления, выносятся в сервисы. Это, к примеру, загрузка данных с сервера, валидация данных, фоновые процессы, логирование. Angular поддерживает DI: сервисы внедряются в компоненты. Для этого используется аннотация @Injectable.

### **2. Инициализация Spring Beans**

Инициализация. Разные способы применяются в таком порядке:

• Метод бина с аннотацией @PostConstruct из стандарта JSR-250 (рекомендуемый способ);

• Метод afterPropertiesSet() бина реализующего интерфейс InitializingBean;

• Init-метод. Для отдельного бина его имя устанавливается в параметре определения initMethod. В xml-конфигурации можно установить для всех бинов сразу, с помощью default-init-method;

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.

### **3. Написать интерфейс для ввода данных банковской карты на React**

import React from 'react';

export function App() {

const [name, setName] = React.useState('');

const [number, setNumber] = React.useState('');

function handleSubmit(event) {

event.preventDefault();

console.log('name:', name);

console.log('number:', number);

}

return (

<form onSubmit={handleSubmit}>

<div>

<label>Name</label>

<input

type="text"

maxLength={5}

value={name}

onChange={(e) => setName(e.target.value)}

/>

</div>

<div>

<label>CVS Number</label>

<input

type="text"

maxLength={5}

value={number}

onChange={(e) => setNumber(e.target.value)}

/>

</div>

<button type="submit">Submit</button>

</form>

);

}

ДОП

**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.

**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 каждыйкласс

Плюсы: возможность null ограничений

Минусы: плохая поддержка полиморфных записей, не нормализованная таблица

3. Своя таблица на каждыйподкласс

Плюсы: нормализованная таблица, возможность null ограничений

Минусы: низкая производительность

**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.

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 Enity 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

**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.

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

Ключевые особенности:

● Таблицы БД описываются в XML-файле, либо с помощью аннотаций.

● 2 способа написания запросов — HQL и Criteria API.

● Есть возможность написания native SQL запросов.

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

**EclipseLink ORM**

ORM-фреймворк от Eclipse Foundation.

Ключевые особенности:

● Основан на кодовой базе Oracle TopLink.

● Является эталонной реализацией (reference implementation) для JPA.

**Связи между сущностями в JPA**

С помощью аннотации @Entity мы указывает 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

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.

**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 начинает 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.

**JPA архитектура**

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

Таблица:

EntityManagerFactory - Это фабричный класс EntityManager. Он создает несколько экземпляров EntityManager и управляет ими.

EntityManager - Это интерфейс, он управляет операциями сохранения объектов. Он работает как factory для экземпляра запроса.

Entity - Сущности - это объекты постоянства, которые хранятся в виде записей в базе данных.

EntityTransaction - Он имеет однозначную связь с EntityManager . Для каждого EntityManager операции поддерживаются классом EntityTransaction .

Persistence - Этот класс содержит статические методы для получения экземпляра EntityManagerFactory.

Query - Этот интерфейс реализован каждым поставщиком JPA для получения реляционных объектов, соответствующих критериям.

**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

**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.

**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.

**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 придётся 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.

**Session Beans, виды и жизненный цикл**

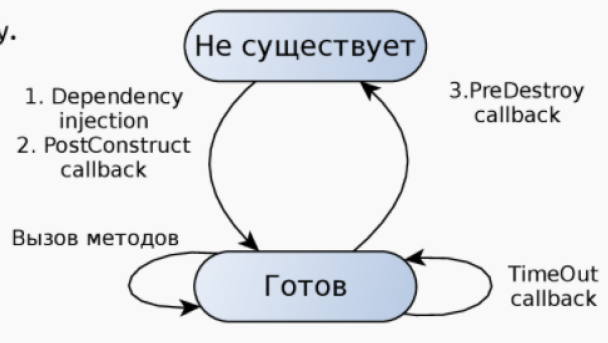
Sessions Beans - предназначены для синхронной обработки вызовов.

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:

Изображение выглядит как текст, снимок экрана, Шрифт, диаграмма

Автоматически созданное описание

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

**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.

**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

**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 Задает 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.

**JPA vs JDBC**

Основное отличия 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).

**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 IoС 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");

JMS и реализация в Java

Позволяет организовать асинхронный обмен сообщениями между компонентами.

Две модели доставки сообщений - «подписка» (topic) и «очередь» (queue):

Изображение выглядит как текст, снимок экрана, диаграмма, линия

Автоматически созданное описание

Изображение выглядит как текст, снимок экрана, линия, Шрифт

Автоматически созданное описание

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, ...)

**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' }" />

**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) ;

}

**Redux Storage, описывающее состояние CRUD интерфейса к списку студентов**

import {

CREATE,

READ,

UPDATE,

DELETE } from 'constants'

const initialState = {

//массив студентов

students = []

}

export function students Reducer(state = initialState, action){

switch(action.type){

case CREATE:

case

return […state, state.students: […state.students, action.payload]}

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"}

}

}

**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>

**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 обосрались");

}

}

}

**Создать бин, конфигурируемый аннотациями, с именем myBean, контекст которого равен контексту другого бина – myOtherBean**

@ManagedBean(name="SampleBean")

public class SampleBean implements Serializable {

@ManagedProperty(value="#{AnotherSampleBean}")

private AnotherSampleBean anotherSampleBean;

}

**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

}

**Criteria Api. Выбрать всех студентов, у которых балл ниже 4 и удалить**

@Stateless

@LocalBean

public class Student Management {

@PersistenceContext

private EntityManager em;

…

public void deletestudent() {

CriteriaBuilder cb = em.getCriteriaBuilder();

CriteriaDelete<Student> delete = cb.createCriteriaDelete(Student.class);

Root e = delete.from(Student.class);

delete.where(cb.lessThan(e.get("avg Grade"), 4.0));

this.em.createQuery(delete).executeUpdate();

}

}

**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="O", minute="O", second="0")

public void midnightMethod() {

em.createQuery("SELECT \* FROM PERSONS")

getResultList ()

.forEach (p->System.out.println(p.name));

}

}

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

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

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 getMilliseconds After Restart() {

return (System.currentTimeMillis() - startTime) / 1000 / 60;

}

public void setMillisecondsAfterRestart()

}

**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();}

}

}

}

**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>