Расписанные билеты по вебу 2023

Билет #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

Invoke Application(Обработка приложения) — На данном этапе JSF обрабатывает события, вызванные нажатием кнопок, ссылок и т.д. Присваивает данные модели. Также решаются вопросы навигации(если ее затрагивали).

##Под капотом##

Вызывается метод UIViewRoot#processApplication() для обработки событий. Перехватить обработку можно переопределением ActionListener

Render Response(Формирование ответа) — JSF генерирует ответ, используя данные, полученные на предыдущих шагах. Обновляет представление страницы, используя managed bean, после чего генерируется html страница, которая отправляется пользователю. Так же, если на предыдущих шага происходили ошибки, то они собираются в тег <messages>.

подробнее тут

- **Способы задания конфигурации Spring**

Самый древний 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.

Второй способ это конфигурирование спринга с помощью джава кода. Для этого нужно создать класс с аннотацией Configuration, и в нем, путем добавления методов с различными аннотациями(bean например) конфигурировать спринг.

Так же спринг приложение можно конфигурировать только аннотациями, добавляя к класса аннотации Component, Bean и т.д. Внедрение зависимостей происходит через аннотацию Autowired.

подробнее тут

- **Написать исходный код CDI бина, реализующего паттерн «команда»**

```
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:

- **IoC и CDI в Java EE**

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.

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

IoC и CDI реализованы в JavaEE (хорошо реализованы с 6-ой версии).

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.

- **Vue.js структура и архитектура: компоненты, экземпляры, директивы**

Vue.js — JS фреймворк, для разработки UI. Использует архитектуру MVVM.

MVVM — Паттерн разработки, позволяющий разделить приложение на три функциональные части:

- Model - основная логика программы (работа с данными, вычислениями, запросы и т.д.)
- View - вид или представление (пользовательский интерфейс)
- ViewModel - модель представления, которая служит прослойкой между View и Model

Экземпляр:

В любом приложении есть хотя бы 1(центральный) экземпляр.

Экземпляр привязывается к узлу DOM, с помощью свойства el.

У экземпляра есть:

$data — Объект с данными

$props — Объект, содержащий текущие входные параметры, которые получил компонент.

$el — элемент, которым управляет экземпляр

$methods (тут понятно)

и другие по типу рутов, парентов и тд.

Компонент:

расширяют функциональность экземпляров

не привязываются к узлам html, а используют собственную разметку, из-за чего их очень удобно переиспользовать.

Директивы:

Позволяют выполнять операции, например итерирование по массиву или включение элементов по условию.

В разметке представляют собой атрибуты тегов.

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

```
	<html
xmlns:f”http://...
xmlns:h”http://...
xmlns:ui”http://...>
	<f:view>
<h:head>
	...
		<title>JSF PAGE</title>
	...
<h:head>
<h:body>
...
	<ui:repeat value”#{news.list} var=”news”>
		<h:outputText value=”#{news.title”} />
<h:outputText value=”#{news.author”} />
<h:graphicImage library="#{news.photo}" name="img.png"/>
<h:outputText styleClass=”aboba” value=”Раскрыть новость”/>
<h:outputText styleClass=”hid text” value=”#{news.text} />
	</ui:repeat>
	...
</h:body>
</f:view>
</html>
```

Билет #3:

- **MVC-модель JSF**

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.

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.

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.

- **Основные аннотации Java EE CDI**

У CDI Bean’ов есть множество аннотаций:

Request/Application/Session/… — аннотация показывает скоуп бина, будет ли он виден только в реквесте или во всем приложении и т.д.

Named — указывает на имя бина

Qualifier — указывает на квалификатор бина

Informal — “бин-наследник”

Так же есть аннотации для методов и полей бина, например

Inject — говорит о том, что конструктор внедрит это значение

и т.д.

- **Написать React компонент формирующий таблицу пользователей, данные приходят в props**

```
function UsersTable(props) {
return (
        {
props.data  && <table className=”table”>
                <thead>
                <tr><td>Name</td><td>Surname</td>...</tr>
                </thead>
                <tbody>
                {
                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.

- **Angular ключевые особенности и различия с AngularJS**

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 контроллеров, их сложно использовать повторно, в Angular’е есть иерархия компонентов, что дает возможно легко и просто реюзать их.

AngularJS универсален, но гораздо менее безопасен и управляем.

AngularJS юзает JavaScript, Angular 2 же использует TypeScript (а значит меньше кода, выше безопасность и ваще язык современнее).

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

- **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:

```
<html
```

```
xlmn:"http://..."
xlmn:h "https://..."
xlmn:f "https://..."
xlmn:ui "https://..."
>
<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:

- **Профили платформы Java EE**

В JavaEE существует два профиля:

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 и еще очень много всего.

- **Типы DI в Spring**

В спринге внедрять зависимости можно двумя способами:

Через конструктор, для этого внедряемый класс должен быть помечен как компонент и его внедрение должно быть либо показано аннотацией autowired, либо прописано в конфигах

Через сеттеры для этого внедряемый класс также должен быть компонентом и его внедрение должно также сопровождаться аннотацией autowired перед сеттером, либо прописано в конфигах

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

```
const router = createBrowserRouter([
        {
        path: “/home”,
        element: (
                <h1>Home</h1>
                <a href=”/news”>News</a>
        )
        },
        {
                path: “/news”,
                element: (
                        <h1>News<h1>
                        <a href=”/home”>Home</a>
                )
        }
])
ReactDOM.createRoot(document.getElementById(“root”)).render
(
```

<RouterProvider router={router}/>

)

Билет #6:

- **Структура JSF 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.

- **Spring MVC: особенности, интеграция в Spring**

Spring Framework — мощнейший фреймворк джавы, разделен на модули, которые удобно дополняют друг друга, но их также можно использовать по отдельности.

В спринге мощная автоматизация почти всех процессов,от внедрения зависимостей, до подключения и работы с базами данных.

Спринг реализует IoC и CDI, позволяет конфигурировать приложения через файлы, аннотации, классы или вообще почти автоматически через SpringBoot.

Spring MVC — модуль спринга, содержащий компоненты, реализующие MVC, ServletDispatcher, ViewResolver, нужные аннотации для контроллеров и бинов, интерфейсы и классы контекстов. В общем все для работы MVC.

- **Написать CDI Bean калькулятор, поддерживающий 4 базовые операции для целых чисел**

@Named(value="calc")

@ApplicationScoped

```
public class Calc {
  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; }
}
```

P.S. правильнее было бы вынести в интерфейс.

Билет #7:

- **Технология RMI. Использование RMI в Java EE**

RMI позволяет клиентскому хосту (ClientHost) вызвать метод, находящийся на сервере (ServerHost), причем вызвать так, как будто метод — локальный.

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.

Подробнее тут

- **Управление состоянием в React. Flux & Redux**

В реакте есть State, который управляет изменением элемента, когда State меняется компонент рендерится повторно. Данное состояние применимо лишь, для того компонента, в котором объявлено.

Так же есть состояние приложения, которое общее для всех страниц(например, чтобы ник юзера был отображен на всех страницах). Этим занимается Redux, который хранит состояния в ReduxStore. Это состояние может менять только метод Dispatch, который принимает Reducer (спец функция, которая принимает старое состояние, действие и тип действия)

- **Напишите пример JSF бина, который при нажатии на кнопку инкрементирует значение на ней**

```
@Named(value=”myBean”)
@ApplicationScoped
public class myBean {
 private int value= 0;
 public void increment() {value++;}
 public int getValue() {return value}
}
```

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

Билет #8:

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

JNDI —API для работы со службой каталогов, Служба каталогов это типа иерархической БД, внутри находятся каталоги(как папки)

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");
```

Плюсы очевидны, мы можем быстро получить доступ к ресурсам по стандартизированной схеме и она безопасна.

Из минусов хз, нашел только что в JNDI+Log4j есть уязвимости.

- **React. Особенности. Архитектура**

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

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

- **Написать бин который выводит количество минут, прошедших со старта сервера**

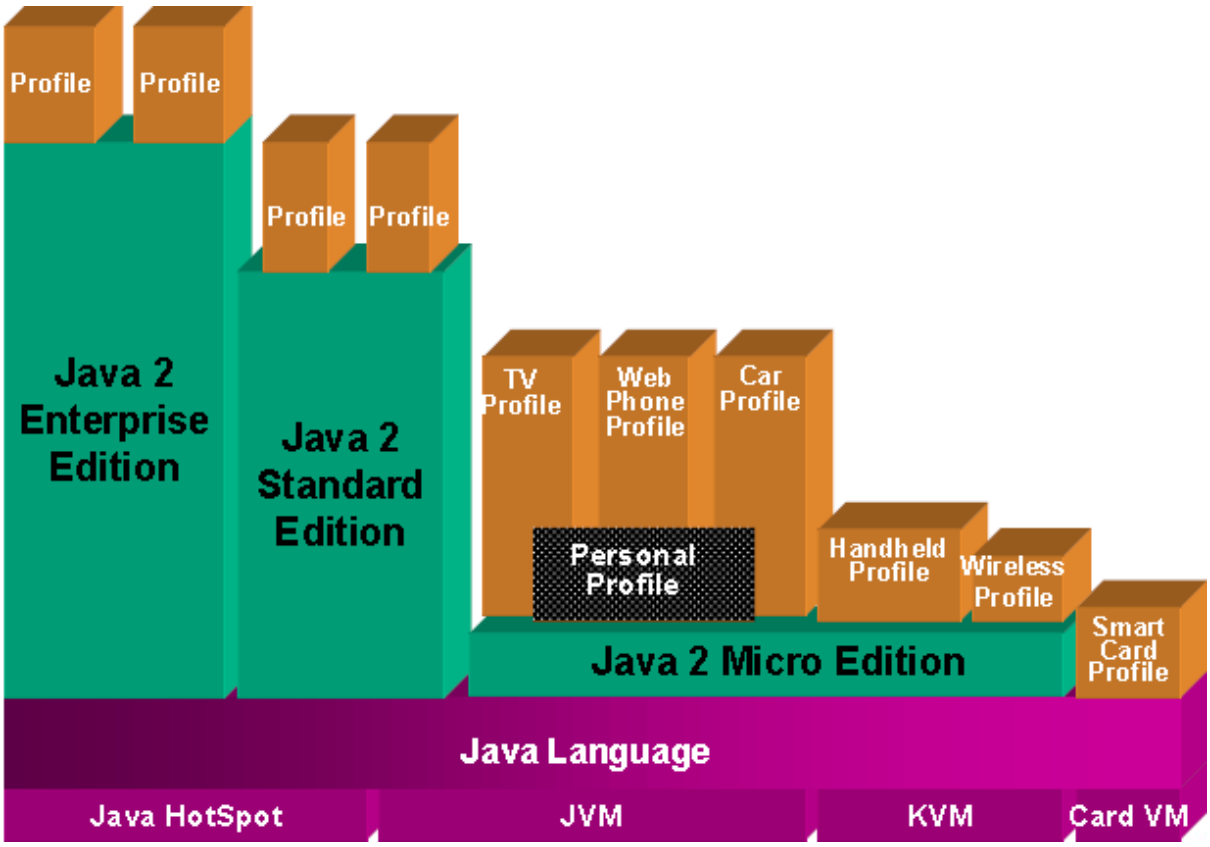
```
@Named(value = "timeHandler")
@ApplicaitonScoped
public class TimeBean {
        private long start;

        public TimeBean() { this.start = System.currentTimeMillis(); }
        public long getTime() { return System.currentTimeMillis() - start; }
        public long setTime() { }
}
```

Билет #9:

- **Платформа Java. Отличия, сходства и что они значат (Java mobile, java TV и тд)**

Есть множество платформ джавы, каждая из которых заточена на решение определенной задачи. Так например Java MicroEdition, нужна для простейших программ, которые можно запускать хоть на калькуляторе, Java SE для небольших приложений, Java EE для крупных проектов, с мощными классами. Так же есть Java mobile (для кнопочных телефонов), Java Android (отдельная джава, со своей структурой и иерархией классов), JavaTV (для SmartTV) и т.д.

Все платформы отличаются наполнение, какие-то дополняют друг друга, какие-то находятся обособленно.



- **Двухфазные и трехфазные конструкторы в Spring и Java EE**

Есть 3 типа конструкторов:

1. Обычный java конструктор
2. Двухфазовый — обычный конструктор + метод с аннотацией PostConstruct. Сначала вызовется обычный конструктор, а затем помеченный метод. 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.

3. Реализаций его нет, нужно ручками писать. Использовать “прокси”. Типа определять поведение объекта до вызова метода. Но для этого нужно наследоваться от BeanPostProcessor’а, и там уже прописывать логику.

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

```
<html
xlmn:”http//...
xlmn:h”http//...
xlmn:f”http//...
xlmn:ui”http//...>
<f:view>
<h:head>
        ...
        <title>Prime numbers</title>
        ...
</h:head>
<h:body>
  <h:commandButton action=”#{nums.generate}” id=”btn”>
    <f:ajax execute=”btn” render=”numbers” />
  </h:commandButton>
        <ui:repeat value=”#{nums.values}” var=”num” id=”numbers”>
                <p>num</p>
        </ui:repeat>
</h:body>
</f:view>
</html>
```

Билет #10:

- **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() {}
```

Так же существуют стандарнтные стереотипы, например @Model (@RequestScoped + @Named)

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

В общем в реакте существует свой формат файлов, JSX. Это надстройка над JS, которая позволяет писать внутри на html'е. В ней можно юзать как обычные html теги, так и кастомные React компоненты.

Пример JSX файла:

```
const [acitve, setActive] = useState(false)
return (
    <button onClick={ setActive(true) }>tap</button>
    <div class=(active ? "" : "hidden")></div>
)
```

- **JSF бин, который извлекает содержимое «такой-то» таблицы базы данных при создании сессии (@SessionScoped)**

```
@Named(value = "dataBean")
@SessionScoped
public class DataBean {
    private List<String> data = new ArrayList();
    @Resource(name="link", type=DataSource.class)
    private DataSource db;

    @PostConstruct
    private void loadData() {
        var connection = db.createStatement();
        var resSet = connection.executeQuery("SELECT *...");
```

```
            while(resSet.hasNext()) {
                data.add(resSet.getString("column");
            }
        }

        public List<String> getData() { return data; }
}
```

Билет #11:

- **Location Transparency в Java EE**

ака Принцип прозрачного нахождения

Благодаря этому принципу, при использовании CDI можно добиться того, что клиенту станет не важно, где физически находится компонент, к которому он обращается (объекты могут находится на разных серверах, но клиент будет думать что приложение — целостное).

Принцип позволяет одинаково обращаться к локальным и удалённым объектам.

Для реализации в Java EE необходимо реализовать RMI: stub (

заглушки) на стороне клиента, и skeleton на стороне сервера.

- **Spring MVC часть представления**

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.

- **Написать managed bean и задать ему scope такой же как у бина otherBean**

@ManagedBean

@ApplicationScoped

public class MyBean {}

@MangedBean

@ApplicationScoped

public class OtherMyBean {}

Билет #12:

- **Валидаторы в JSF. Создание, назначение и тд.**

в JSF доступны 4 вида валидации:

1. С помощью встроенных компонентов (DoubleRangeValidator, LengthValidator и др.)

   Пример:
   <h:inputText id=... size=...>

   <f:validateLongRange minimum=”0” maximum=”100”/>

</h:inputText>

2. На уровне приложения

Эти валидаторы мы пишем сами, методы добавляются в управляемые бины

3. Если у нас тип данных, для которого нет стандартного валидатора, мы можем создать свой на стороне сервера.

Пример:

public void validateMe(FacesContext context, UIComponent component, Object value) throws ValidatorException {...}

<... validator=”#{object.validateMe}” ...>

4. С помощью специальных компонентов, реализующих интерфейс Validator. JSF позволяет создавать собственные компоненты для валидации. Для этого нужно создать класс, реализующий

интерфейс Validator, и реализовать метод validate(). Затем, нужно зарегистрировать класс в faces-config.xml. Потом можно использовать тег в вёрстке.

- **Реализация контроллера в Spring MVC**

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

- **Vue.js простейший чат бот, который на любое сообщение отвечает «сам дурак»**

html:

```
<div id=”example”>
        <input placeholder=”введите сообщение”></input>
        <p ref=”ref”></p>
        <button v-on:click=”answer”>Отправить</button>
</div>
```

js:

```
let example = new Vue({
        el: ‘#example’,
        …
methods:{
        answer:  function(){
                this.$refs.ref.innerText = “Сам дурак.”;
}
}
})
```

Билет #13:

- **Фаза формирования представления JSF**

Фаза формирования представления (RestoreView), на этой фазе фреймворк(JSF) делает следующее:

- Если это первое обращение клиента, то создаются объекты, назначаются слушатели, валидаторы, конверторы, все это помещается в FacesContext и отправляется на создание ответа юзеру.
- Если FacesContext уже есть (это не первый запрос и все объекты уже созданы), то проверяется тип запроса, а после запускаются процессы рендера новых страниц(если это GET) и создание/изменение данных(если это POST).

- **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, в спринге нужно писать руками.

- **REST-контроллер, реализующий конвертацию температуры из Цельсия в Фаренгейты и наоборот**

```
@RestController
@RequestMapping(“api/temp”)
public class TempController() {
	@PostMapping(“/cels”)
	public double getCels(@RequestVariable(“fars”) doule fars) {
		// Algorithm fars-to-cels
	}
```

```
    @PostMapping("/fars")
    public double getFars(@RequestVariable("cels") doule cels) {
        // Algorithm cels-to-fars
    }
}
```

Билет #14:

- **Java EE CDI Beans прерывание жизненного цикла (Interception)**

Interceptors позволяют перехватывать вызовы методов бизнес-логики и жизненного цикла CDI бинов. Для этого создается кастомная аннотация, которой помечаются методы. Interceptors можно использовать для логирования, управления транзакциями, ограничения времени выполнения метода.

Для создания Interceptor:

Создаем свою аннотацию, помеченную @InterceptorBinding

Создаем свой класс перехватчик, помечаем его нашей аннотацией и @Interceptor. Метод перехватчика, помеченный аннотацией @AroundInvoke будет вызываться при входе и выходе из перехваченного метода.

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

React реализует подход переиспользования компонентов. Мы можем создавать компоненты, а потом переиспользовать их на страницах (кнопки, навбары, футеры, формы и т.д.)

Создать компонент можно 2-мя способами: Создать класс, расширяющий React.Component, либо создать функцию, возвращающую View. Умные компоненты хранят в себе некое состояние и могут меняться в зависимости от него. Глупые ничего не хранят и просто выводят что-то на экран.

Умные компоненты умеют делать запросы, обрабаывать инфу и т.д.

Глупые же сконцентрированы на UI, и в основном используют Props, для реюзабильности.

У каждого копмонента есть свои State и Props. Они нужны для отслеживания изменения какого-нидудь состояния или для гибкости(чтобы передавать инфу от компонента в компонент)

- **Написать JSF страницу с текстовыми полями принимающие латинские символы**

```
<html ...>
...
<h:body>
    <h:inputTextArea>
        <f:validateRange pattern=”[a-z, A-Z]*”/>
    </h:inputTextArea>
</h:body>
...
</html>
```

Билет #15:

- **Rest контроллер в Spring: сериализация и десериализация данных**

Про модели я уже писал.

Когда запрос попадает в контроллер, пройдя перед этим валидацию тела и заголовков, начинается его десериализация. Происходит это посредством различных аннотаций, таких как @RequestParam, @PathParam и прочих, или с помощью различных дешифрующих библиотек. Так мы достаём данные из запроса.

После получения ответа от модели, контроллер его сериализует для отправки обратно. Следовательно, необходимо “свернуть” данные в нужный формат (JSON или XML чаще всего). С этим помогают библиотеки Jackson, JAXB и некоторые другие.

- **Архитектура приложения в Ангуляр: модули, компоненты, представления, сервисы**

Наше приложение состоит из нескольких модулей, которые можно import друг в друга. В каждом модуле есть компоненты, которые связаны с представлением. Компонент отвечает непосредственно за логику представления, и связан с ним двусторонней связью – при изменении поля на странице меняется переменная в компоненте, и наоборот. Есть ещё сервисы, которые отвечают за логику, не связанную с представлением. Компоненты и сервисы можно инжектить в другие компоненты и сервисы.

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

```
<faces-config version="2.2"
...
>
  <managed-bean>
        <managed-bean-name>myBean</managed-bean-name>
        <managed-bean-class>MyBeanClass</managed-bean-class>
<managed-bean-scope>#{CUSTOM_SCOPE}</managed-bean-scope>
        <managed-property>
              ...
        </managed-property>
</faces-config>
```

И отдельно необходимо создать класс, который будет наследоваться от ConcurrentHashMap<String, Object>, в котором программист уже будет реализовывать управление бином. (Также, в классе нужно будет объявить константу SCOPE = “CUSTOM_SCOPE” чтобы была возможность использовать этот класс как свой scope).

Билет #16:

- **JSF: ключевые особенности, преимущества, недостатки**

JSF реализует чёткое разделение бизнес-логики и представления, приложения на JSF построены по шаблону MVC, где Model - Managed Beans, View - JSP или XHTML страница, а Controller - FacesServlet, который уже есть и нам не надо его писать.

Преимущества: управление обменом данных на уровне компонент, простая работа с событиями, возможность подключать наборы компонентов, широкая поддержка от IDE.

Недостатки: сложно реализовывать непредусмотренную функциональность, сложно писать свои компоненты.

- **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 позволяет даёт возможность управлять bean-компонентами. В отличие от MB, CDI бины намного мощнее и гибче, они могут использовать перехватчики,

стереотипы, декораторы и многое другое. EJB же обладают некоторыми особенностями, недоступными для CDI (например, транзакционные функции). Однако, в целом, EJB и CDI схожи, и их можно даже инжектить друг в друга.

- **Angular компонент, который позволяет поделиться чем-то в VK, Twitter, Facebook (API для соцсетей можно описать текстом)**

```
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);
}
}
```

API принимает логин и пароль, а также сообщение, которое будет опубликовано на личной странице после прохождения авторизации. Ненавижу Angular.

Билет #17:

- **Создание пользовательского интерфейса в JSF-приложениях. Иерархия компонентов JSF.**

Интерфейс JSF строится из компонентов. Структура представляет из себя дерево, с корнем в UIViewRoot.

Компоненты расположены на JSP страницах, все компоненты реализуют интерфейс UIComponent.

На компоненты можно ставить слушатели, валидаторы, конвертеры, правила навигации. Так же благодаря xml верстке, мы можем расширять набор компонентов, используя сторонние библиотеки, либо вообще написав свои.

- **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.

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

- **Написать приложение на angular, содержащее форму с именем и датой, которое в ответ генерирует заполненный бланк псж**

```
@Component({
	template:
<form #form=”ngForm”>
	<label>Имя</label>
	<input name=”name” [(ngModel)]=”name” />
	<label>Фамилия</label>
	<input name=”sur” [(ngModel)]=”sur” />
	<button> (click)=”send()”>
Отправить
</button>
		</form>
})

export class AppComponent {
	name: string=””;
	sur: string=””;

	send() {
		// Отправка формы
	}
}
```

Билет #18:

- **Managed beans, конфигурация, скоупы**

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. Скоупы бывают:

RequestScoped - инстанс бина будет создаваться каждый HTTP запрос.

SessionScoped - бин будет доступен на всё время сессии, то есть инстанс будет создан 1 раз при подключении клиента

ViewScoped - создаётся один раз при обращении к странице

Application scoped - создаётся один раз при запуске сервера и живёт, пока сервер не выключат

CustomScoped - мы сами можем управлять этим скоупом. Для этого бин надо поместить в map тогда, когда он станет нужен

NoneScoped - бин не привязывается ни к одной из областей. Можно применять к вспомогательным бинам.

Настраивать бины (в том числе и их скоупы) можно либо через аннотации (@ManagedBean, @SessionScoped), либо через xml файл faces-config: там заполняется имя, класс, scope а также managed property нужного бина. Но легче, конечно, через аннотации)

- **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’;
import Cookies from ‘js-cookie’;
```

```
class App extends React.Component {
        this.state = {
        jSessionid: 0
}
getJSessionid(){
        fetch("Некий URL нашего API", {
        method: "GET",
        ...
}).then(res=>this.setState(jSessionid: res.headers.get('jSessionid')))
```

} - это вообще нужно? Из условия не понятно, дан ли нам изначально этот jSessionid, или нужно пришвартовываться к серверу для его получения -_-

```
componentDidMount(){
getJSessionid();
}

render(){
        let form;
        if(Cookies.get('this.state.jSessionid')){
                form = <и тут форма для логина и пароля, ага />;
}
}

return (
        <div>
                {form}
        </div>
)
}
```

Билет #19:

- **Контекст управляемых бинов. Конфигурация контекста бина**

было в прошлом билете

- **Шаблоны MVVM и MVP. Сходства и отличия от MVC**

MVP, или Model-View-Presenter - шаблон, созданный много позже MVC. С развитием средств веб-программирования отпала необходимость выносить в отдельную категорию контроллер, однако появился Презентер. В него входит то, что отвечает за отрисовку и обновление View, а также то, что отвечает за обновление Model. Простыми словами, он содержит логику интерфейса и отвечает за синхронизацию Model и 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. Этот шаблон рекомендуется использовать в проектах, где реализовано связывание данных (DataBinding) – прямая связь между представлением и моделью без использования контроллера или презентера. Сама же ViewModel содержит модель, преобразованную к представлению, а также команды, через которые представление обращается к модели.

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

```
import React from "react";


function CompletionBox(props){
        return props.list.map((word) => <li>{word}</li>);
}


function Autocomplete(props){
getList(event){
str = event.target.value;
fetch('https://api.npms.io/v2/search?q='+str)
.then(response => response.json())
.then(data => setState({list = data;}));
}
render() {
return(
 <div>
  <input onChange={this.getList} />
        <CompletionBox list={this.state.list}/>
```

```
 </div>)
}
}

export default Autocomplete;
```

Билет #20:

- **REST в спринге: методы и аргументы**

REST в спринге реализовано при помощи REST контроллеров

(либо обычных контроллеров, но тогда к методам нужно добавлять ResponseBody).

Есть аннотации упрощающие жизнь(например @RequestBody в параметре конвертит тело запроса в параметр). Также методы REST контроллеров возвращают объекты конвертированные в json, для этого есть класс ResponseEntity(внутри реализация на Jakson).

- **Навигация в реакте. React Router**

React Router — апи для навигации по приложению. Т.к. сам реакт реализует подход SPA(одностраничное приложение), а нам хочется передвигаться по нему по url'ам роутер очень нужная вещь.

Его работа основана на прослушивании апи браузера, а именно стека истории, туда он пушит, попит и реплейсит страницы, перерендеривая интерфейс.

- **Реализовать бронь авиабилетов на jsf**

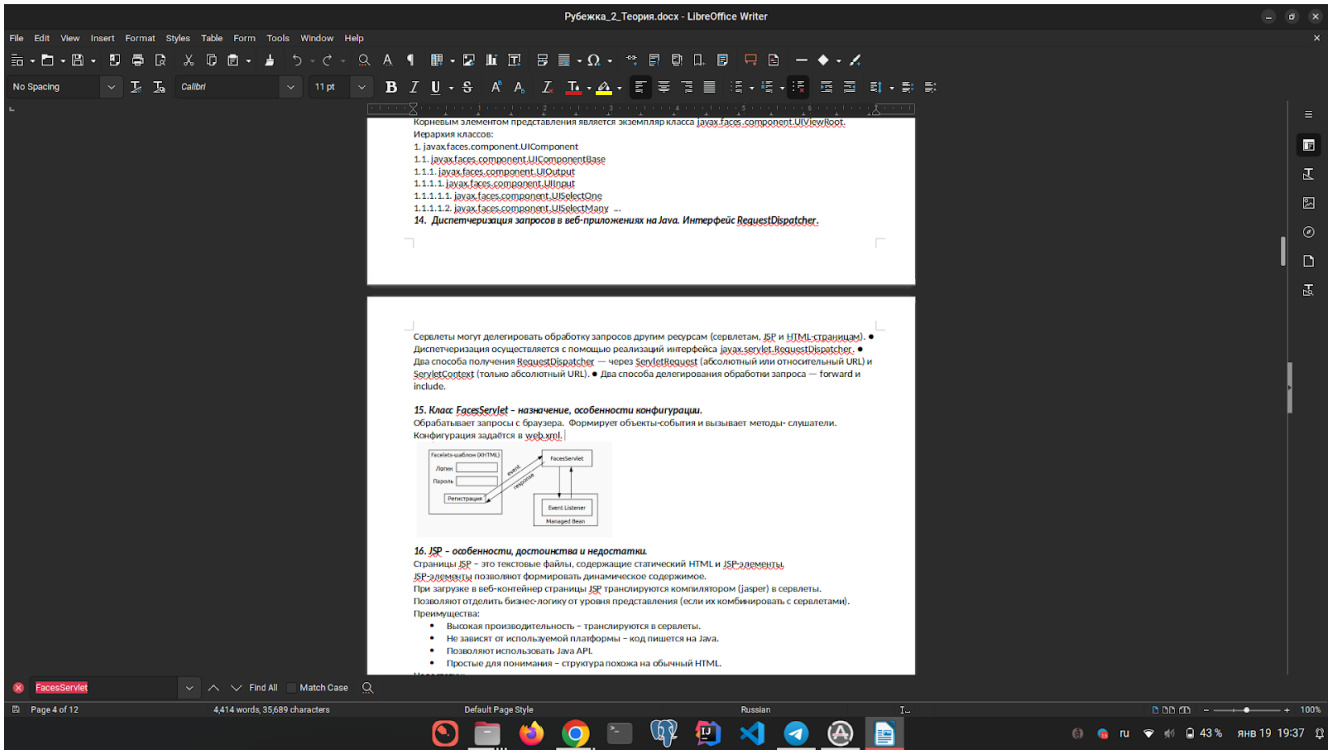
Ну тут снова пишем верстку, которая получает массив свободных билетов, туда кнопку забронировать, и пусть она посылает запрос с номером билета на серв, а сервер удаляет его из массива|бд|чего-угодно

Билет #21:

- **Класс FacesServlet - назначение, особенности конфигурации**

FacesServlet - главный сервлет в JSF проекте, он первым инициализируется при старте приложения, и в дальнейшем управляет жизненным циклом обработки запросов. А именно ожидает запросы, запускает методы по валидациям, слушателям, заполнениям бинов и генерациям ответов.

Этот сервлет конфигурируется в файле web.xml, (в целом как обычный сервелет с классом jakatra.faces.webapp.FacesServlet



- **Vue.js - ключевые особенности, преимущества и недостатки**

Vue.js — фреймворк для создания пользовательского интерфейса и SPA на JS (или TS). Реализует паттерн MVVM и является компоненто-ориентированным.

Компоненты Vue чаще всего пишутся в одном файле, в котором совмещается код, шаблон и стили компонента. Поддерживается one-way и two-way data-binding, то есть при изменении свойств меняется страница, и наоборот, при изменении данных на странице (например через поле для ввода) меняются свойства компонента. Компоненты образуют иерархию, передавая дочерним компонентам свойства (props, так же как в React).

Плюсы:

- высокая производительность за счет Virtual DOM

- маленький размер бандла

- реактивность

- лаконичность, читаемость кода

Минусы:

- не поддерживается мобильная разработка
- экосистема не так развита, как у React
- меньше разработчиков, знакомых с Vue

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

```
@RequestMapping(“/api/pokemons”)
@RestController
public class PokemonController {
	@GetMapping()
	public List<Pokemon> show() {
		return pokRes.findAll();
	}

	@GetMapping(“/{id}”)
	public Pokemon showById(PathVariable(“id”) long id) {
		return pokRes.findById(id);
	}

	@PostMapping()
	public void save(RequestParam(“pokemon” Pokemon pokemon) {
		pokRes.save(pokemon);
	}

	@PostMapping(“/{id}”)
	public Pokemon showById(RequestParam(“id”) id, ReqPar(“pok”) Pok pok {
		return pokRes.replace(id, pok);
	}
```

// + DeleteMapping по айди

// + еще можно добавить патч, но вообще постзапрос по айди должен заменять сущность(зависит от договоренностей, насколько я знаю)

}

Билет #22:

- **Конверторы JSF, создание и назначение**

Конвертеры данных используются в компонентах для преобразования данных в заданный формат. Для основных типов данных уже существуют стандартные конвертеры, которые можно подключить специальным тегом (например, <convertDateTime/>. Создать собственный конвертер можно реализовав интерфейс Converter в своём классе. Необходимо будет реализовать методы getAsObject() (для преобразования ввода в объект) и getAsString, а затем зарегистрировать конвертер в faces-config.xml или пометить его аннотацией @FacesConverter(name).

- **Реализация model в Spring web MVC**

Моделью в Spring MVC является совокупность POJO, или бинов. Они выполняют основную бизнес-логику программы и достаются контроллером по необходимости. 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.

- **Написать на vue.js интерфейс для навигации по текстовому документу, представленному в виде большой строки, должны быть реализованы переходы на следующую и предыдущую страницу**

html:

<input type="button" v-on:change="next">Следующая страница</input>

<input type="button" v-on:change="prev">Предыдущая страница</input>

{{ pages[num] }}

js:

```
let example = new Vue({
        el: '#example',
data: {
                num: 0,
                pages: [],
                document: document //наш документ
        },
        created: function() { //функция вызовется один раз перед отрисовкой
                const length = 1000; //будем выводить по тысяче символов на страницу
                const pattern = new RegExp(".{1," + length + "}", "ig");
                this.state.pages = this.document.match(pattern).map(item => item.padEnd(length, "."));
}
methods:{
        next:  function(){
                inc this.state.num;
},
prev: function(){
        dec this.state.num;
}
}
})
```

Билет #23:

- **CDI Beans**

уже блин было кучу раз

- **Angular DI**

Иногда нам требуется использовать компонент или сервис в другом компоненте/сервисе. Здесь и появляется Dependency Injection: помечаем нужный класс аннотацией @Injectable и далее, в конструкторе компонента, в котором нам потребуется нужный класс, встраиваем его.

Пример:

```
@Injectable()
class Aboba {
}

class Amogus {
        constructor(private aboba: Aboba){ }
        ...
}
```

- **JSF страничка с данными из бина**

Уже было 100 раз

Билет #24:

- **Шаблоны и представление в Angular**

Представление в Angular задаётся с помощью шаблонов (template). Компонент может содержать несколько уровней представлений, в том числе представления из других встроенных компонентов. Шаблон очень похож на классический HTML, а взаимодействие с компонентом происходит с помощью ссылок на его свойства ({{aboba.name}} - пример ссылки на свойство aboba).

Шаблон и компонент связаны двусторонне - мы можем влиять на шаблон из компонента и наоборот. Так, например, в шаблоне можно указать ссылку на метод компонента, который выполнится после исполнения какого-нибудь действия в шаблоне.

- **Dependency Lookup Spring**

В Спринге есть два вида внедрения зависимостей: Dependency Injection и Dependency Lookup. В DI всё просто - там спринг делает всё сам, мы лишь помечаем аннотацией @Autowired то “место”, куда нужно предоставить нужный бин. А Спринг сам чешет контейнер и достаёт то, что нужно. Но, иногда, DI может дать сбой (достать не тот бин, например). Тогда мы можем сами поискать нужную зависимость.

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.

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

Нужно добавить следующее в web.xml:

<**servlet-mapping**>

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

<**url-pattern**>/faces/*.xhtml</**url-pattern**>

</**servlet-mapping**>

Билет #25:

- **Process validations phase, Update module values phase**

Выполнение проверок, на этой фазе все строковые значения преобразуются в объекты. Те, объекты что еще не прошли валидацию проходят ее тут. Для этого вызывается метод UIViewRoot#processValidators().

На этапе Update Model Values можно гарантировать, что все объекты отвалидированы и проверены, после этого все объекты с “дерева” переносятся в модель, тем самым данные на сервере обновляются. Для этого (как ни странно) вызывается метод UIVIewRoot#processUpdates().

- **Жизненный цикл Spring-приложения**

1. BeanDefinitionReader читает файлы конфигурации и создаёт мета информацию о бинах (Bean Definition, для каждого бина свой)
2. Создаются beanFactoryPostprocessor’ы, которые могут поменять дефинишены в зависимости от того, как они настроены
3. BeanFactory читает дефинишены, вызывает конструкторы бинов, затем отдает бины постпроцессорам, которые парсят аннотации и тд.
4. BeanFactory отдаёт бины в IoC контейнер. Приложение готово к работе.

- **Интерфейс на Angular, который выводит интерактивные часы с обновление каждую секунду**

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


@Component({
  selector: 'my-app',
  template: <div class="form-group">
                                <div class=”time”>{{ time }}</div>
        </div>
})


export class Time {
                time: “”;
                setInterval(() => {
                        const now = new Date();
                        this.time = now;
}, 1000);
```

Билет #26:

- **Реализация Ajax в jsf**

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

Старый: с помощью JavaScript API jsf.ajax.request(). Обеспечивает стандартный мост для запросов и детальный контроль. Пример: <h:commandButton ... onClick=”jsf.ajax.request(this, event, {execute:’myinput’, render:’outtext’’}); return false;”/>

Новый: тегом <f:ajax>. Очень удобно, не нужно трогать JavaScript. Пример: <h:commandButton>

<f:ajax execute=”@form” render=”outtext”/>

</h:commandButton>

В обоих случаях атрибут render отвечает за перерисовку одного из компонентов страницы, чтобы отразить внесенные изменения.

- **CDI beans: контекст (Bean Scope)**

было.

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

```
function SignInForm() {

const [login, setLogin] = useState(“”);
const [password, setPassword] = useState(“”);

function signIn() {
        const URL = “/api/sign/in”

        ( async () => {
                let data = new FormData();
                data.append(“username”, login);
                data.append(“password, password);

                let result = await fetch(URL, {method: “POST”, body: data});
                let message = result.json();

                // Логика проверки, успешно ли мы вошли и тд
        })()
}

return (
<form onSubmit=”{signIn”}>
        <input placeholder=”login” onChange={(e) => setLogin(e.target.value); }/>
        <input placeholder=”password” onChange={(e) => setPassword(e.target.value); }/>
</form>
)
}
```