JSF

JSF структура

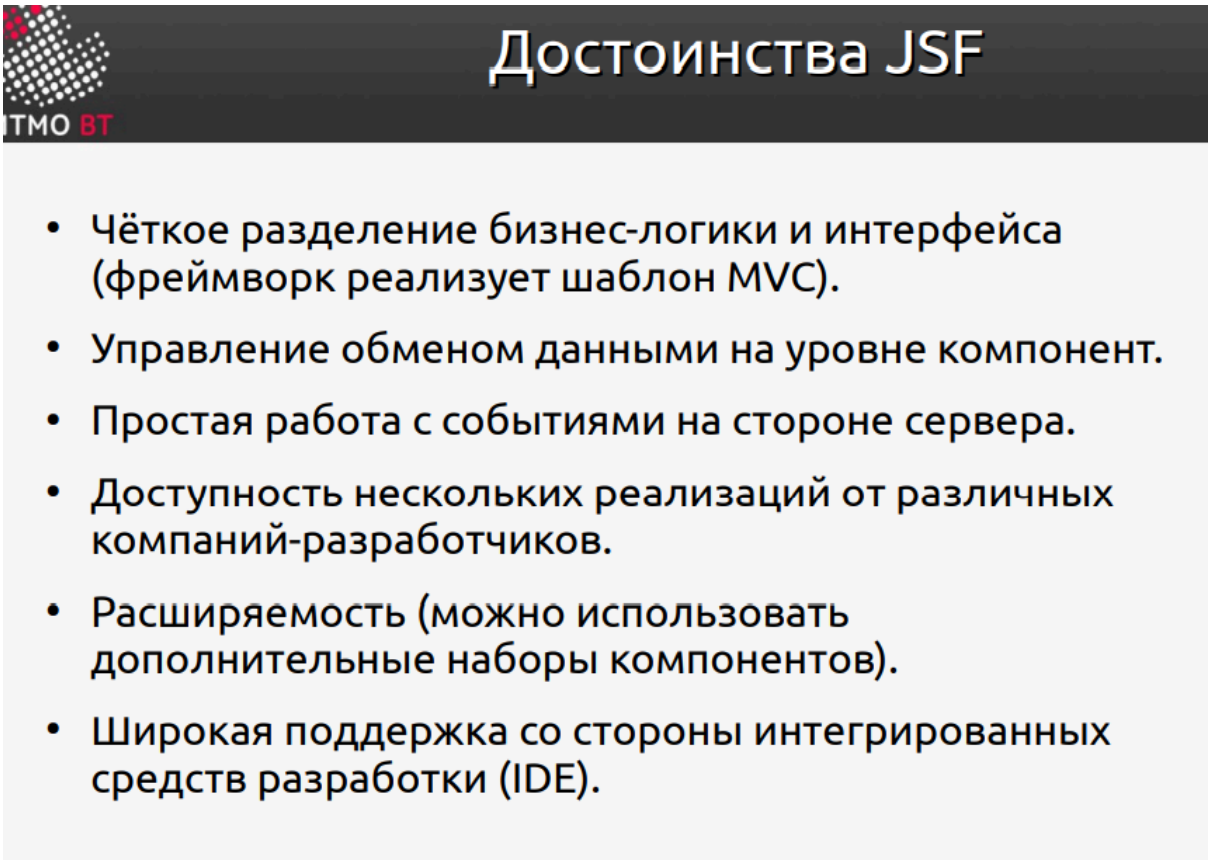
JSF реализует MVC модель, которая разделяет уровень представления с уровнем бизнес логики. В качестве контроллера выступает специальный FacesServlet, представление - Facelet или JSP-страница, модель - набор управляемых бинов, реализующих логику на стороне сервера.

Структура JSF-приложения

- JSP-страницы с компонентами GUI

ние

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



Недостатки JSF

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

Managed Beans

Управляемые бины - это сериализуемые классы, которые нужны для того, чтобы содержать в них данные и логику обработки этих данных. Бины подключаются через faces-config.xml или аннотации в программу, после чего можно доставать их поля (имеющие геттеры и сеттеры) через EL прям из JSF страницы. Методы управляемых бинов можно ставить в качестве обработчиков событий тех или иных ивентов на странице JSF.

У бинов есть область жизни - контекст - он же scope (про него след пункт)

- 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.

JSF. Контекст manage бинов

У бинов есть область жизни, явно задаваемая конфигом:

- **@RequestScoped -** используется по умолчанию. Создаётся новый инстанс *managed bean* на каждый HTTP запрос. Если, например форма будет содержать данные, которые необходимо будет отправить на сервер для обработки, то инстанс данного бина будет создаваться 2 раза: 1 - создаётся по первому запросу (*initial request*), 2 - создаётся по отправке формы (*postback*). Контекст —запрос.
- **@SessionScoped -** инстанс создаётся один раз при обращении пользователя к приложению, и используется на протяжении жизни

сессии.*Managed bean* обязательно должен быть Serializable. Контекст — сессия.

- **@ApplicationScoped -** инстанс создаётся один раз при обращении, и используется на протяжении жизни всего приложения. Не должен иметь состояния, а если имеет, то должен синхронизировать доступ, так как доступен для всех пользователей. Контекст — приложение.
- **@ViewScoped -** инстанс создаётся один раз при обращении к странице, и используется ровно столько, сколько пользователь находится на странице (включая ajax запросы). Контекст — страница.
- **@CustomScoped(value="#{someMap}") -** инстанс создаётся и сохраняется в Map. Программист сам управляет областью жизни
- **@NoneScoped -** инстанс создаётся, но не привязывается ни к одной области жизни. Полезно применять в *managed bean*'е, на который ссылаются другие *managed bean*'ы, имеющие область жизни

Про CustomScoped: управляем сами, но как бы нет. Схема такая: положили в мапу - бин создался, убрали из мапы - бин уничтожился

конфигурация с помощью аннотаций или в xml файле

Способ 1 — через faces-config.xml:

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

Способ 2 (JSF 2.0) — с помощью аннотаций:

```
@ManagedBean(name="customer")
@RequestScoped
public class CustomerBean {

  ...
  @ManagedProperty(value="#{initParam.defaultAreaCode}"
    name="areaCode")
  private String areaCode;
  ...
}
```

JSF. FacesServlet. Конфигурация

Обработкой запросов с JSF-страниц занимается специальный класс FacesServlet, потому ему нужно назначить обработку jsf в web.xml:

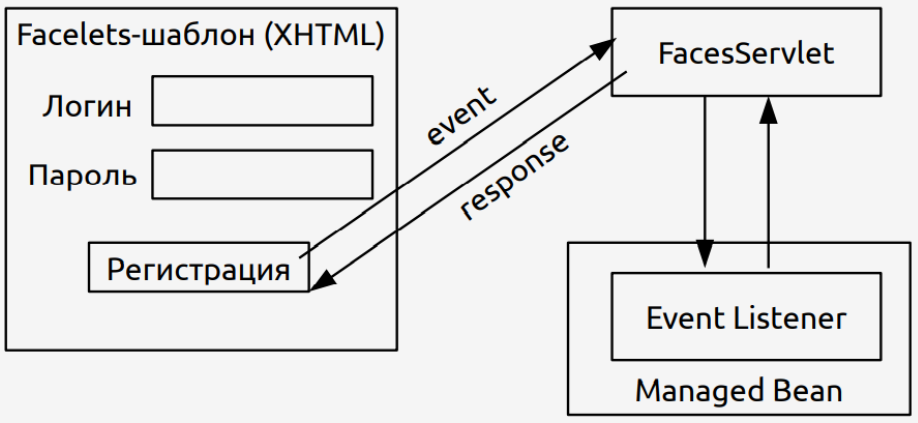
Сервлет, который будет делегировать выполенение запросов на указанных урлах jsf фреймвоку

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

```
<servlet>
    <servlet-name>Faces Servlet</servlet-name>
    <servlet-class>javax.faces.webapp.FacesServlet</servlet-class>
    <load-on-startup>1</load-on-startup>
</servlet>
<servlet-mapping>
    <servlet-name>Faces Servlet</servlet-name>
    <url-pattern>/faces/*</url-pattern>
</servlet-mapping>
```

Собственно, обычный сервлет, у которого также в web.xml нужно расписать мапинги. Только мы физически не создаём класс этого сервлета (это всё делает JSF рантайм)

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



JSF. Валидаторы

Валидация данных JSF-компонентов

- 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.
- Можно создавать собственные валидаторы.

Существует 4 типа валидации

1. С помощью встроенных компонентов
2. На уровне приложения
3. С помощью проверочных методов серверных объектов (inline-валидация)
4. С помощью специализированных компонентов, реализующих интерфейс Validator

1. С помощью встроенных компонентов

1. DoubleRangeValidator
2. LongRangeValidator
3. LengthValidator

```
<%-- возраст (age) --%>
<h:outputLabel value="Age" for="age" accesskey="age" />
<h:inputText id="age" size="3" value="#{contactController.contact.age}">
<f:validateLongRange minimum="0" maximum="150"/>
</h:inputText>
<h:message for="age" errorClass="errorClass" />
```

2. На уровне приложения

Это непосредственно бизнес-логика. Заключается в добавлении в методы управляемых bean-объектов кода, который использует модель приложения для проверки уже помещенных в нее данных.

3. С помощью проверочных методов серверных объектов

Для типов данных, не поддерживаемых стандартными валидаторами, например, адресов электронной почты, можно создавать собственные валидирующие компоненты

ание

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

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

 </h:selectOneRadio>
```

4. С помощью специализированных компонентов, реализующих интерфейс Validator

JSF позволяет создавать подключаемые валидирующие компоненты, которые можно использовать в различных Web-приложениях.

Это должен быть класс, реализующий интерфейс Validator, в котором реализован метод validate(). Необходимо зарегистрировать валидатор в файле faces-config.xml. После этого можно использовать тег <f:validator/> на страницах JSP.

faces-config.xml

```
<validator>
  <validator-id>arcmind.zipCode</validator-id>
  <validator-class>com.arcmind.validators.ZipCodeValidator</validator-class>
</validator>
```

JSF. Конвертеры

Конвертеры данных

ITMO BT

- Используются для преобразования данных компонента в заданный формат (дата, число и т. д.).
- Реализуют интерфейс `javax.faces.convert.Converter`.
- Существуют стандартные конвертеры для основных типов данных.
- Можно создавать собственные конвертеры.

Стандартные конвертеры JSF

- javax.faces.BigDecimal
- javax.faces.BigInteger
- javax.faces.Boolean
- javax.faces.Byte
- javax.faces.Character
- javax.faces.DateTime
- javax.faces.Double
- javax.faces.Float

```
<h:outputLabel value="Age" for="age" accesskey="age" />
<h:inputText id="age" size="3" value="#{contactController.contact.age}">
</h:inputText>

<h:outputLabel value="Birth Date" for="birthDate" accesskey="b" />
<h:inputText id="birthDate" value="#{contactController.contact.birthDate}">
<f:convertDateTime pattern="MM/yyyy"/>
</h:inputText>
```

С пециализированные конвертеры

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

файл faces-config.xml

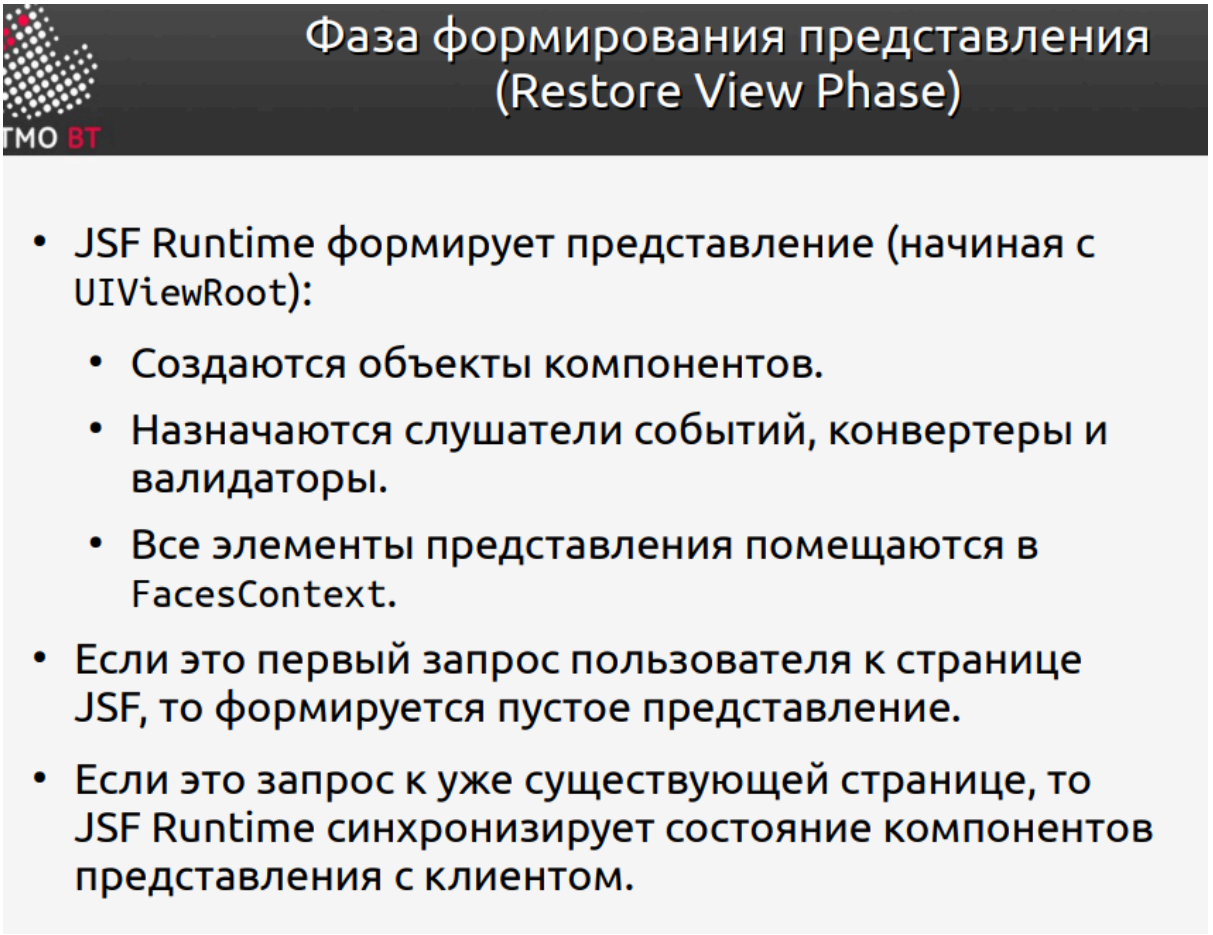
```
<converter>
  <converter-for-class>
    com.arcmind.contact.model.Group
  </converter-for-class>
  <converter-class>
```


ние

```
    com.arcmind.contact.converter.GroupConverter
(com.arcmind.contact.converter.TagConverter)
  </converter-class>
</converter>
```

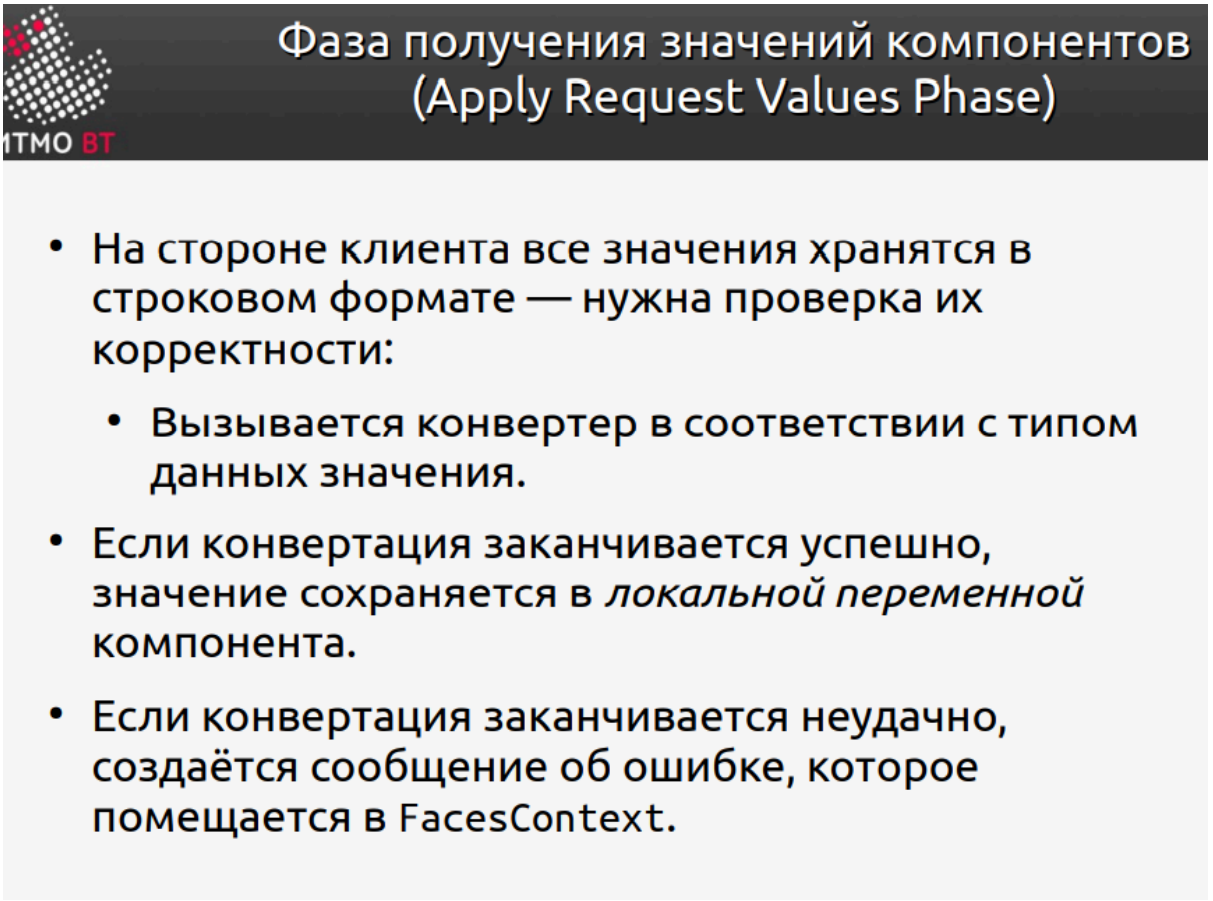
JSF. Жизненный цикл

1 фаза



- Фаза формирования представления. JSF Runtime формирует представление по запросу(request) пользователя: создаются объекты компонентов, назначаются слушатели событий, конвертеры и валидаторы, все элементы представления помещаются в FacesContext

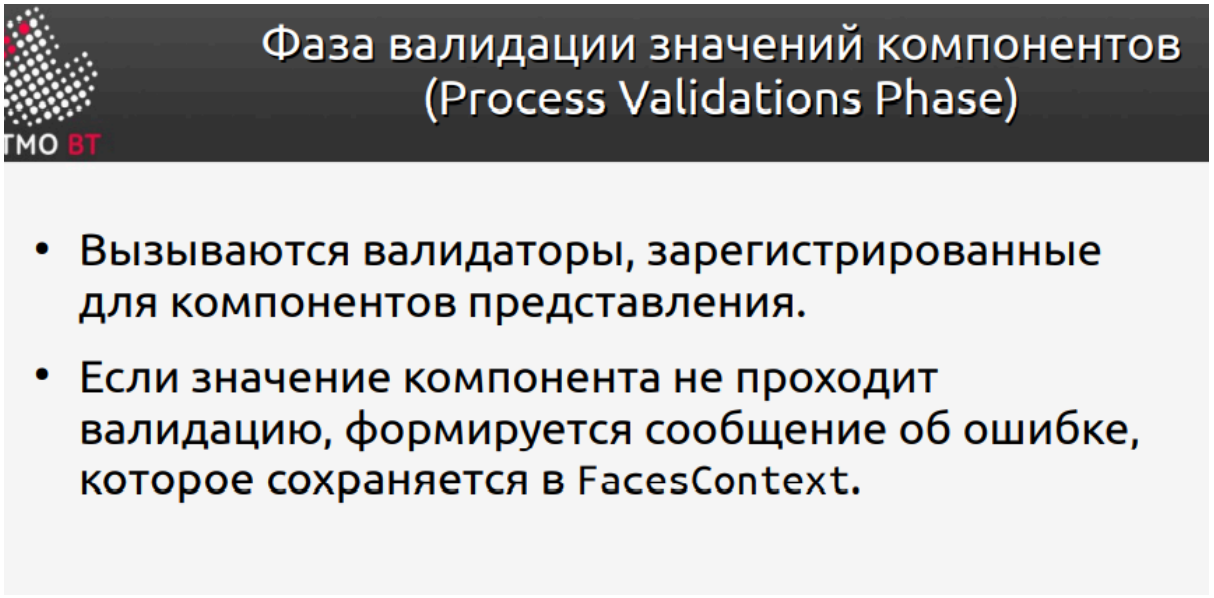
2 фаза



- Фаза получения значений компонентов. Вызывается конвертер из стокового типа данных в требуемый тип. Если конвертация успешна, то значение сохраняется в локальной переменной компонента. Если неуспешно – создается сообщение об ошибке и помещается в FacesContext.

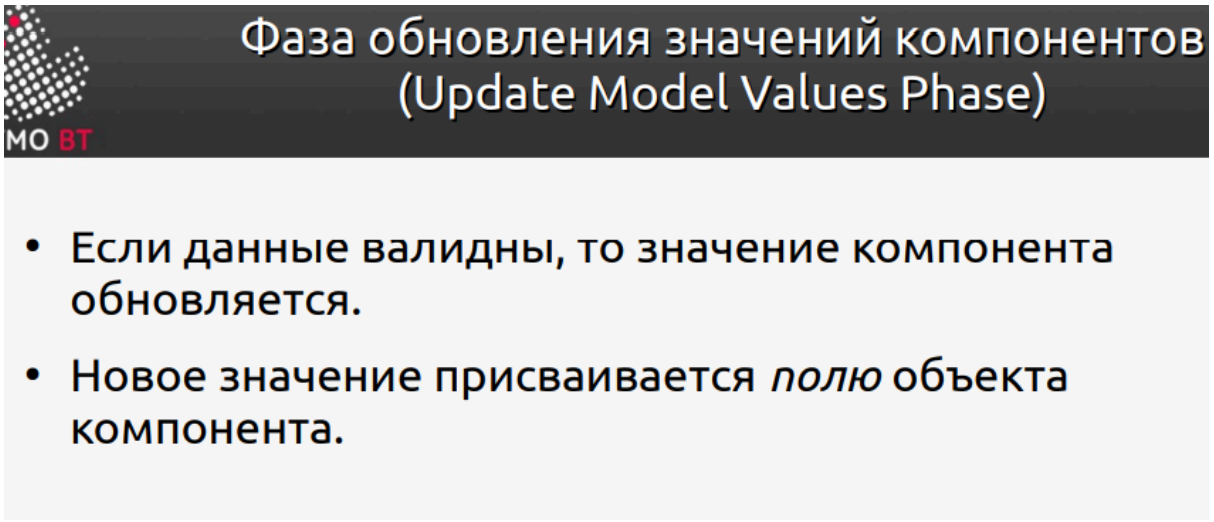
Локальная переменная компонента - это переменная в каком-то методе (компонент - это то, что хранится у нас сервере и реализует UIComponent)
Значение компонента - наш managed bean
Если всё плохо на данном этапе, то переходим к 6ой фазе

3 фаза



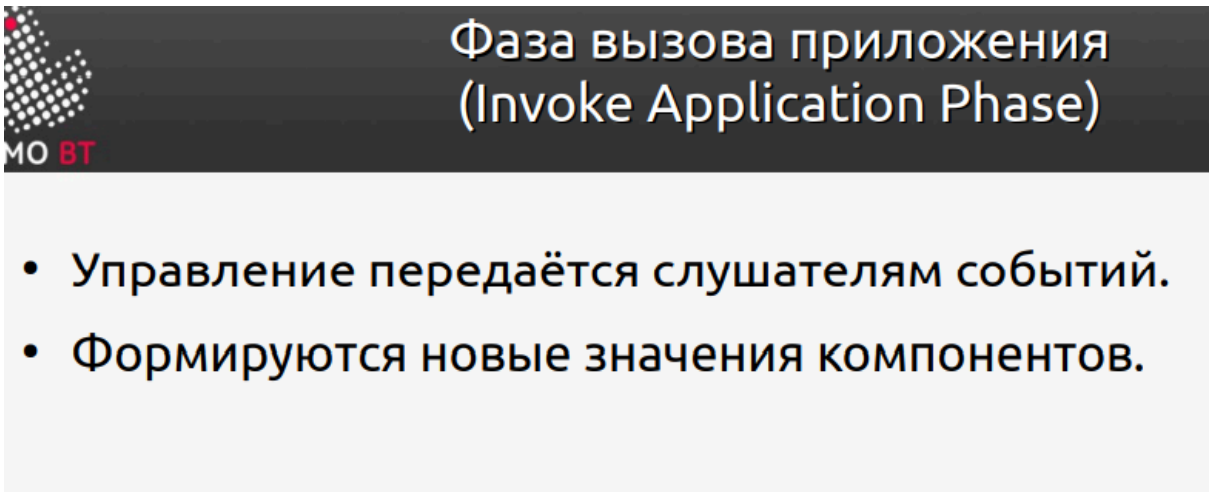
Если всё плохо, то переходим к 6ой фазе (ответ клиенту)

4 фаза



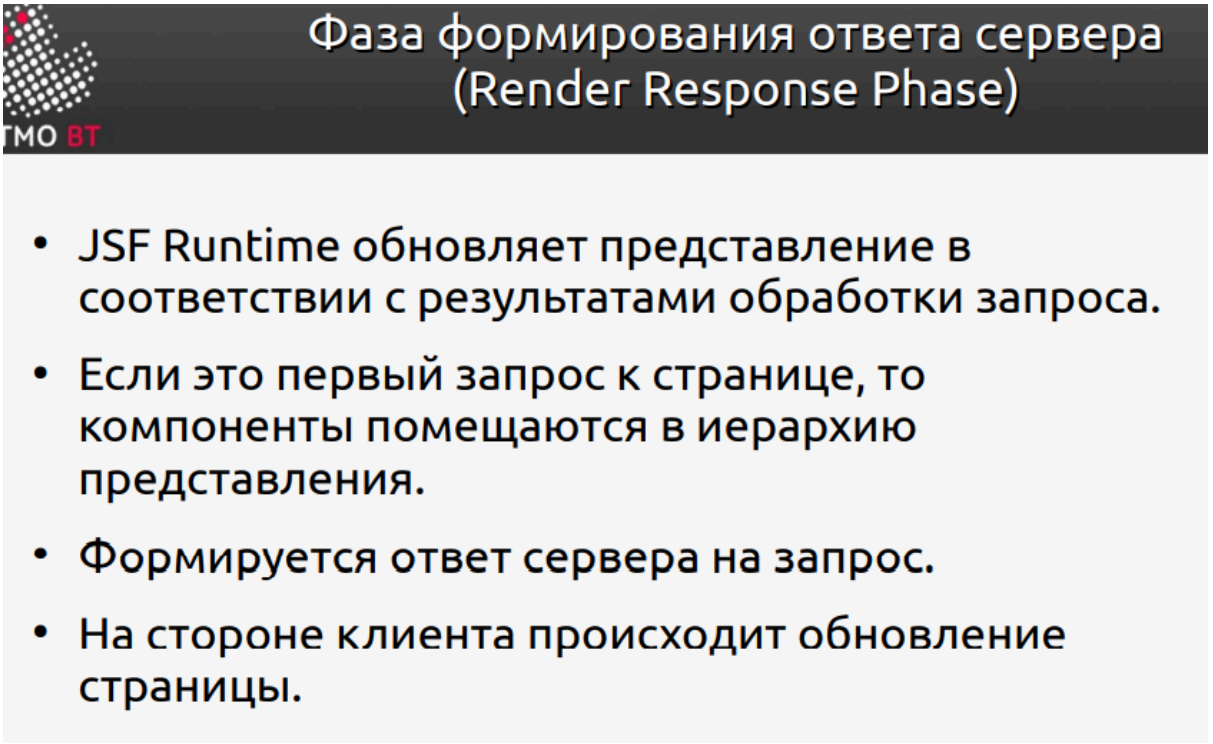
Значение компонента и есть наш бин, собственно обновляется модель

5 фаза



Собственно тут и происходит вся та бизнес-логика, которую написал программист. Конечно, в ходе её мы можем снова обновить значения пропертей бина, тем самым обновив значение компонента

6 фаза



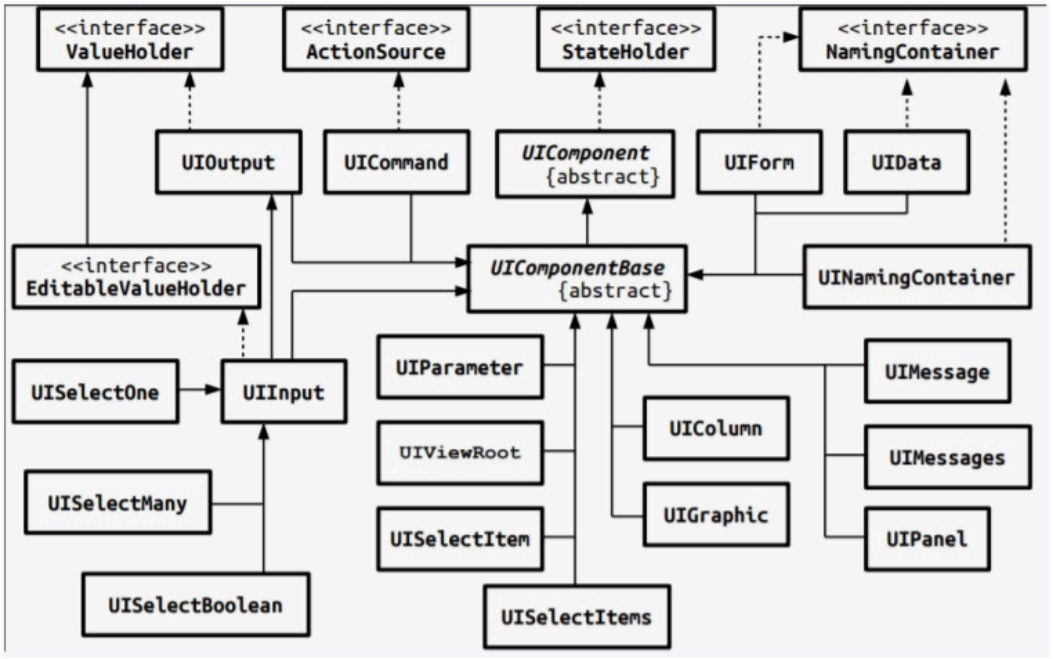
JSF. Компоненты

Особенности реализации JSF-компонент

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

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

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



Дополнительные библиотеки компонентов.

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

JSF. Ajax

Поддержка Ajax в JSF 2 предоставляется двумя основными способами. Первый — это JavaScript API: jsf.ajax.request (). Этот API-интерфейс обеспечивает стандартный мост для запросов Ajax и обеспечивает большой детальный контроль. Второй — это новый тег с именем `<f: ajax>`. С этим тегом вам вообще не нужно беспокоиться о JavaScript. Вместо этого вы можете использовать этот тег для декларативного добавления поведения Ajax в ваше приложение.

Примеры:

```
<h:commandButton id="submit" value="submit"
onclick="jsf.ajax.request(this, event, {execute:'myinput',render:'outtext'}); return false;" />
```

```
<h:commandButton id="submit" value="submit">
  <f:ajax execute=”@form” render=”outtext”/>
</h:commandButton>
```

Собственно render - что изменится
execute - указали форму

JSF. Facelets & JSP

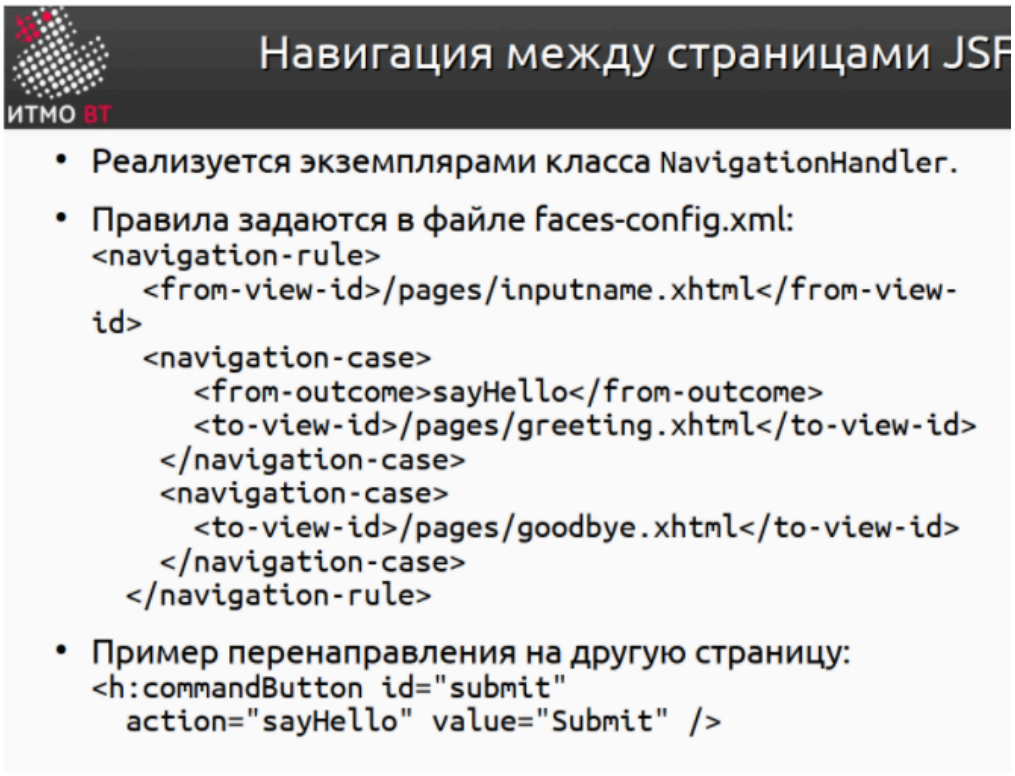
Facelets

- Особенности
  - Для создания страниц используется XHTML (transitional)
  - Использование библиотек тегов (через пространства имен)
  - Поддержка Expression Language
- Преимущества
  - Повторное использование кода (шаблоны и компоненты)
  - Возможность настройки и корректировки работы компонентов
  - Быстрая компиляция
  - Проверка выражений EL на этапе компиляции
  - Быстрый рендеринг компонентов

Интерфейс JSF-приложения состоит из страниц JSP (Java Server Pages), которые содержат компоненты, обеспечивающие функциональность интерфейса. При этом библиотеки тегов JSP используются на JSF-страницах для отрисовки компонентов интерфейса, регистрации обработчиков событий, связывания компонентов с валидаторами и конверторами данных и многого другого.

При этом нельзя сказать, что JSF неразрывно связана с JSP, т.к. теги, используемые на JSP-страницах только отрисовывают компоненты, обращаясь к ним по имени. Жизненный же цикл компонентов JSF не ограничиваетJSP-страницей.

JSF. Навигация



Особо добавить нечего. Просто задаем кнопке, ссылке и т. п. action и отлавливаем его в конфиге, перенаправляя в нужное место.

Ссылку можно добавить тремя различными способами:

- С помощью commandLink и обычного правила перехода, определяемого в faces-config.xml

```
<navigation-rule>
  <navigation-case>
    <from-outcome>CALCULATOR</from-outcome>
    <to-view-id>/pages/calculator.jsp</to-view-id>
  </navigation-case>
</navigation-rule>
```

- С помощью commandLink и правила перехода, использующего элемент .
- Связывание с помощью прямой ссылки (элемента <h:outputLink>)

```
<h:outputLink value="pages/calculator.jsf">
<h:outputText value="Calculator Application (outputlink)"/>
</h:outputLink>
```

RMI

RMI позволяет клиентскому хосту (Client Host) вызвать метод, находящийся на серверном хосте (Server Host). Причем вызывает так, будто он локальный.

(1) Регистрируем серверный объект в RMI Registry

У серверного объекта, чтобы его можно было вызвать, должен быть публичный интерфейс (обычный Javaский интерфейс с набором public методов).

Stub - Заглушка, Skeleton - каркас. Оба формируются рантаймом.

Server Stub - имплементация серверного интерфейса, которая живет на клиентской машине и делает вид, что она и есть серверный объект. При этом серверной логики внутри Stub нет. Запросы, полученные Stub, Stub отправляет на сервер в Server Skeleton, который вызывает методы уже физического объекта. После Skeleton возвращает результат работы метода Stub, а тот передает его клиенту в месте вызова.

За физический удаленный вызов отвечают Server Stub и Server Skeleton.

(2) Поиск объекта сервера

(3) Возращение серверного Stub клиенту

(4) обмен данными

Клиент работает с сервером через интерфейс с серверным Stub, думая, что это сам серверный объект.

JNDI

JNDI - апишка, для работы со службой каталогов. Служба каталогов - некая иерархическая БД. Почему Служба каталогов - каталоги внутри.

JNDI - другой аспект использования службы каталогов. Теперь в иерархическую структуру мы объединяем не студентов, а Java`ские ресурсы. Например, для обеспечения доступа к ManagedBean CalculatorBean из других бинов, мы регистрируем ссылку на CalculatorBean в службе каталогов.

Взаимодействие со службой каталогов в Java происходит через JNDI.

(по слайду:)

Есть Java приложение, которому нужно получить доступ к ресурсу, например, обратиться к ManagedBean. JNDI API позволяет осуществлять поиск в иерархической службе каталогов.

```
// Пример получения ссылки на JDBC datasource.
DataSource dataSource = null;
try {
   // Инициализируем контекст по умолчанию.
   Context context = new InitialContext();
   dataSource =
      (DataSource) context.lookup("Database");
} catch (NamingException e) {
    // Ссылка не найдена
}
```

Location Transparency

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

Location Transparency означает, что при использовании CDI механики мы можем добиться того, что нам станет не важно, где физически находится компонент, к которому мы обращаемся. (он может находиться локально, удаленно, внутри контейнера, вне контейнера). То есть если JNDI мы ищем DataSource, нам не важно, где будет физически находится объект на который нам ссылка в DataSource вернется.

С помощью LocTransp мы можем одинаково обращаться как к локальному объекту, так и удаленному (за ним стоит некоторая инфраструктура). Получением удаленного объекта занимается сервер приложений.

Java EE. Спецификации и их реализации

Java EE - довольно страшный монстр, который нужен не всегда.

В приложениях, ориентированных на несложный веб, не нужны слои бизнес-логики и данных.

Существует только 2 профиля - Web и Full.

Web Profile Java EE приложения - это те приложения, которые обеспечивают работу веб-стека.

В Full Profile поддерживаются все компоненты спецификации Java EE.

Начиная с Java EE 6 Application серверу дозволено быть конструктором, то есть он может включать в себя не все компоненты. Множество компонентов, которое Application сервер в себя включает, называется профилем. Спецификация Java поддерживает сколько угодно профилей, однако по факту существует только один - Web Profile - компоненты для веб-приложений.

не уверен, что здесь про это

Принципы Ioc и CDI

Inversion of Control - принцип при котором созданием и инъекцией наших компонентов занимается не программист, а контейнер. Т.е контейнер управляет жизненным циклом компонентов и отвечает за взаимодействие между ними.

CDI (Context & Dependency injection) - позволяет снизить или совсем убрать зависимость компонента от контейнера.

Компонент, конечно, зависим от контейнера, так как второй его порождает. Мысль в том, что программист не зависит от API, предоставляемого контейнером. API, предоставляемые контейнером используется декларативно.

То есть когда мы пишем компонент, мы обычно не реализуем их интерфейсов, все взаимодействие с контейнером и все спецификации того, что за компонент мы пишем, мы делаем с помощью аннотаций. Мы не вызываем API напрямую. Это тоже обычно делается через аннотации. Получается, что вся логика взаимодействия с контейнером делается через метапрограммирование. То есть мы задаем метаданные о том, что это за компонент, что с ним нужно делать, на какие события жизненного цикла мы вешаем обработчики, и с какими внешними компонентами он взаимодействует и каким образом.

EJB

EJB. Компонентная модель

EJB - Enterprise java Beans. Компоненты, реализующие паттерн "бин", но немного по-другому.

- EJB компоненты инкапсулированы контейнером.

  То есть напрямую мы к ним не обращаемся, а обращаемся с помощью инъекций.

- Контейнер предоставляет клиентам прокси-объекты (proxy objects) для доступа к EJB компонентам.

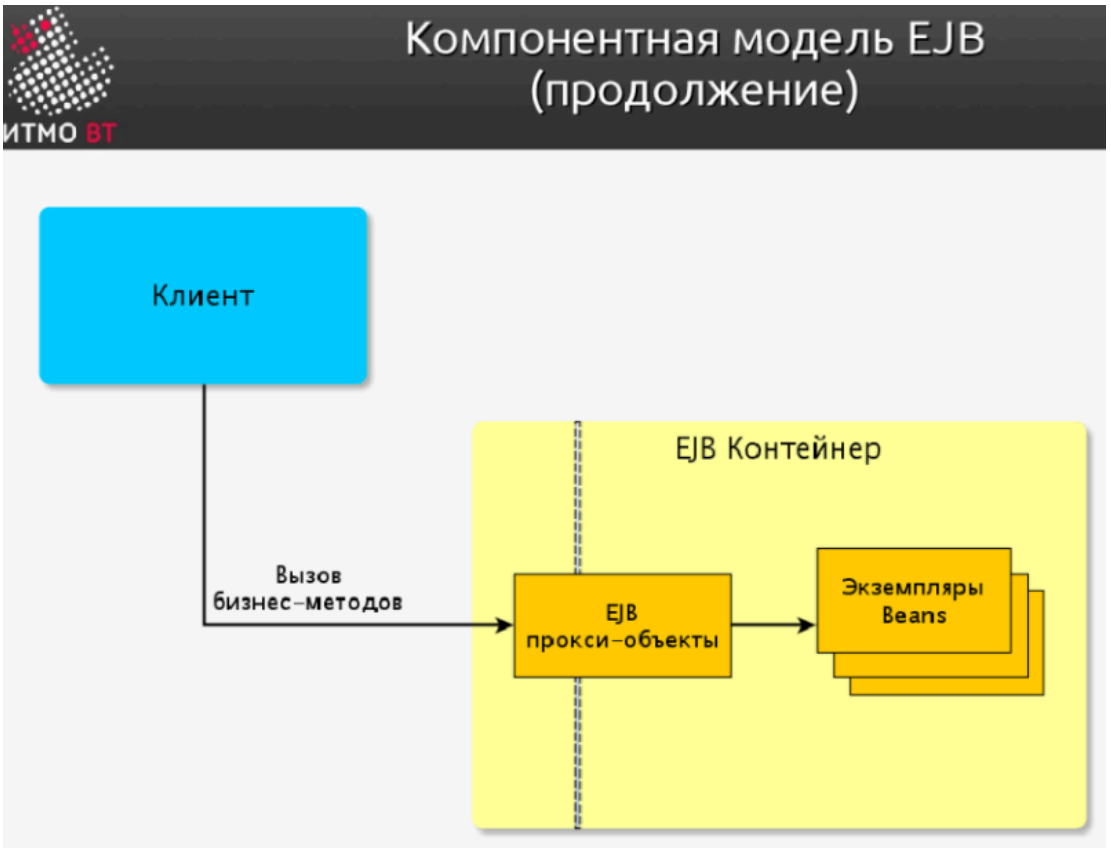
  Схема похоже на схему RMI (со Stub и Skeleton). То есть когда мы вызываем EJB, тс есть говорим: "EJB, сделай что-нибудь", мы вызываем не его, а прокси-объект. Это прокси-объект может быть самим бином (если бин локальный). Прокси-объект генерируется контейнером.

- Прокси-объекты реализуют бизнес-интерфейсы.

  Каждому бину обязаны определить обычный Java`ский интерфейс., который говорит о том, что данный бин должен уметь с точки зрения использующих его сервисов. Все, что находится внутри бина помимо интерфейса, снаружи недоступно. Поэтому прокси-объекты - это сгенерированные контейнером объекты, реализующие бизнес-интерфейс.

- Клиенты вызывают методы бизнес-интерфейса.

  Они не знают, локальный этот бин или удаленный, они обращаются к прокси-объекты через бизнес-интерфейс.



Все, что находится за пунктирной чертой - скрыто от клиента (он не знает, как это работает и ему это не важно).

Прокси объект на основании настроек контейнера будет решать, как передать запрос дальше к реальному бину.

EJB бины

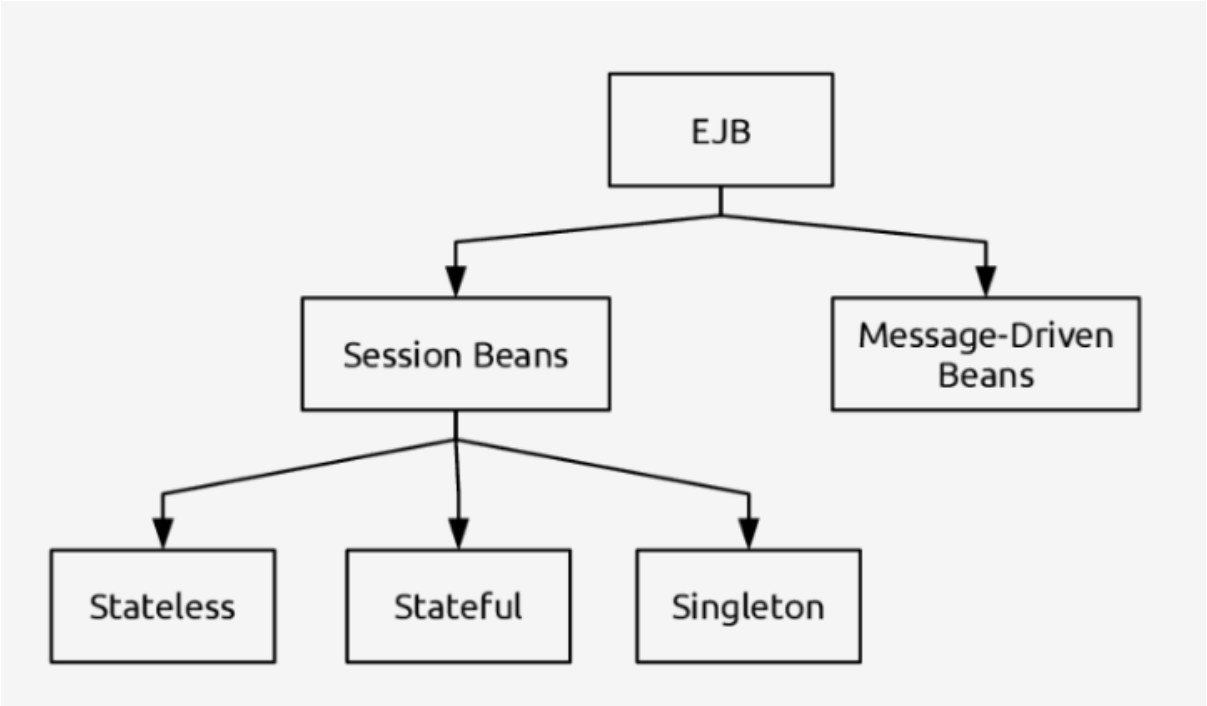
EJB - компоненты Java EE, отвечающие за уровень бизнес-логики

В Java EE Full Profile managed бины отвечают только за логику, связанную с рендерингом интерфейса (перехватывает событие нажатия на кнопку). Вся остальная работа происходит в EJB.

Уровень бизнес-логики в приложениях обычно наиболее тяжелый, поэтому есть смысл его декомпозировать для эффективного масштабирования.

Особенности EJB:

- Возможность локального и удалённого доступа.
- Возможность доступа через JNDI или Dependency Injection.
- Поддержка распределённых транзакций (с помощью JTA).
- Поддержка событий.
- Жизненным циклом управляет EJB-контейнер (в составе сервера приложений).



Обращение к session bean из managed и unmanaged кода

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

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

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

Компоненты EJB. Stateless & Stateful Session Beans. EJB Lite и EJB Full.

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

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

Session beans в свою очередь делятся на:

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

Stateless

Не сохраняют состояние между последовательными обращениями клиента.

Нет привязки к конкретному клиенту.

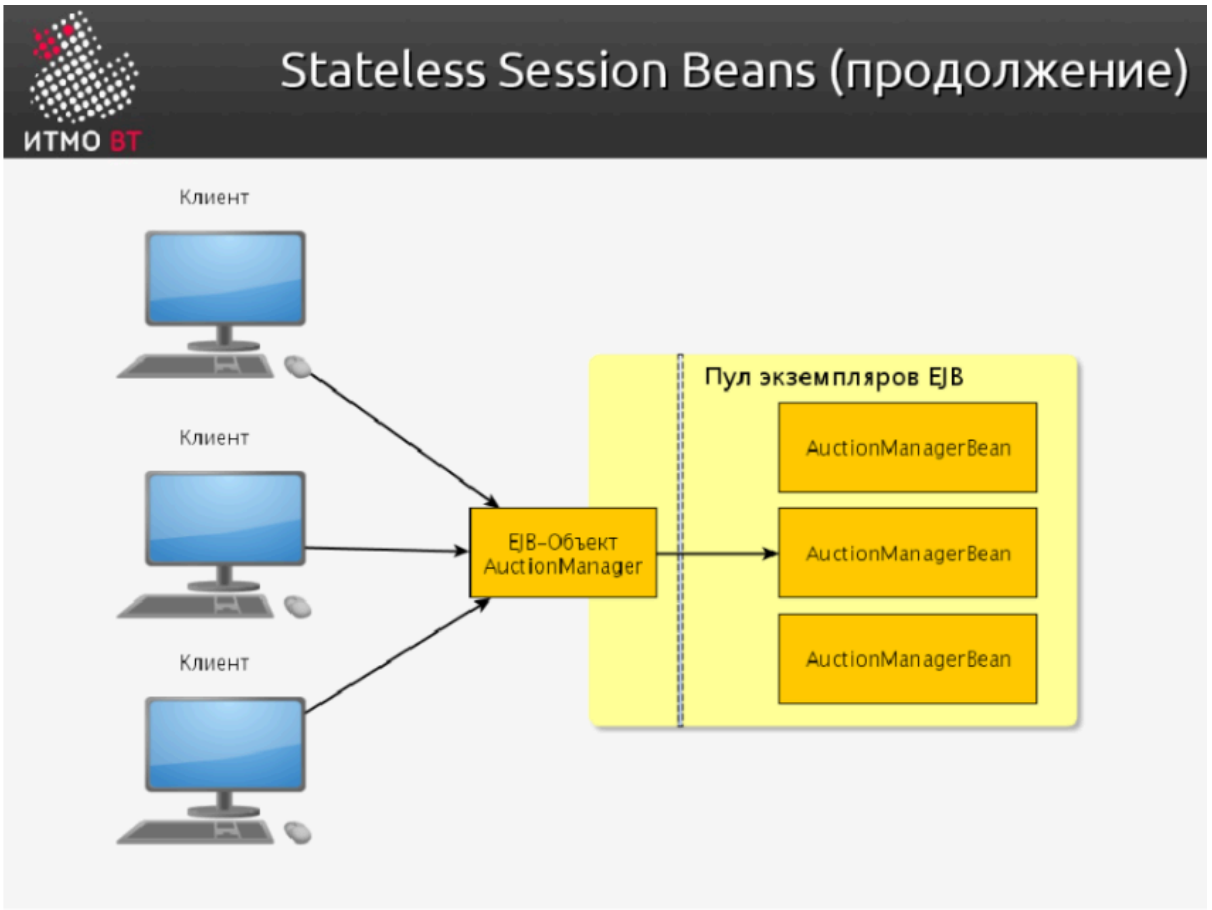
Хорошо масштабируются.

Объявление — аннотация `@Stateless`.

Вертикальное масштабирование - улучшение железа (рост вычислительной мощности конкретной машины)

Горизонтальное масштабирование - добавление доп сервера. Приложение распараллелилось, быстрее работать не стало, но с той же скоростью стало тянуть больше запросов. Возможность горизонтального масштабирования предъявляет определенные требования к архитектуре приложения.

Stateless бины не хранят состояние, то она хорошо масштабируется. Их можно раскидать по разным серверам.



Есть:

- Набор клиентов. Клиентами на самом деле являются другие бины.
- Сформированный настройками сервера пул экземпляров EJB
- EJB-объект, является прокси для взаимодействия с бинами, через него происходит обращение к какому-то из пула экземпляров Stateless бина.

Клиентская часть приложение обратилась к бину, например, заинжектила его. Дальше для обработки запроса клиента прокси-объект на основании алгоритма выбирает один экземпляр из пула бинов. Таким образом, разные запросы от одного клиента скорее всего будут выполняться на разных физических бинах. Поэтому при создании в бине поля и сохранения в нем каких-то данных при запросе, мы будем получать разные значения этого поля, так как бины при запросах будут разные. Поэтому состояние бина использовать нельзя.

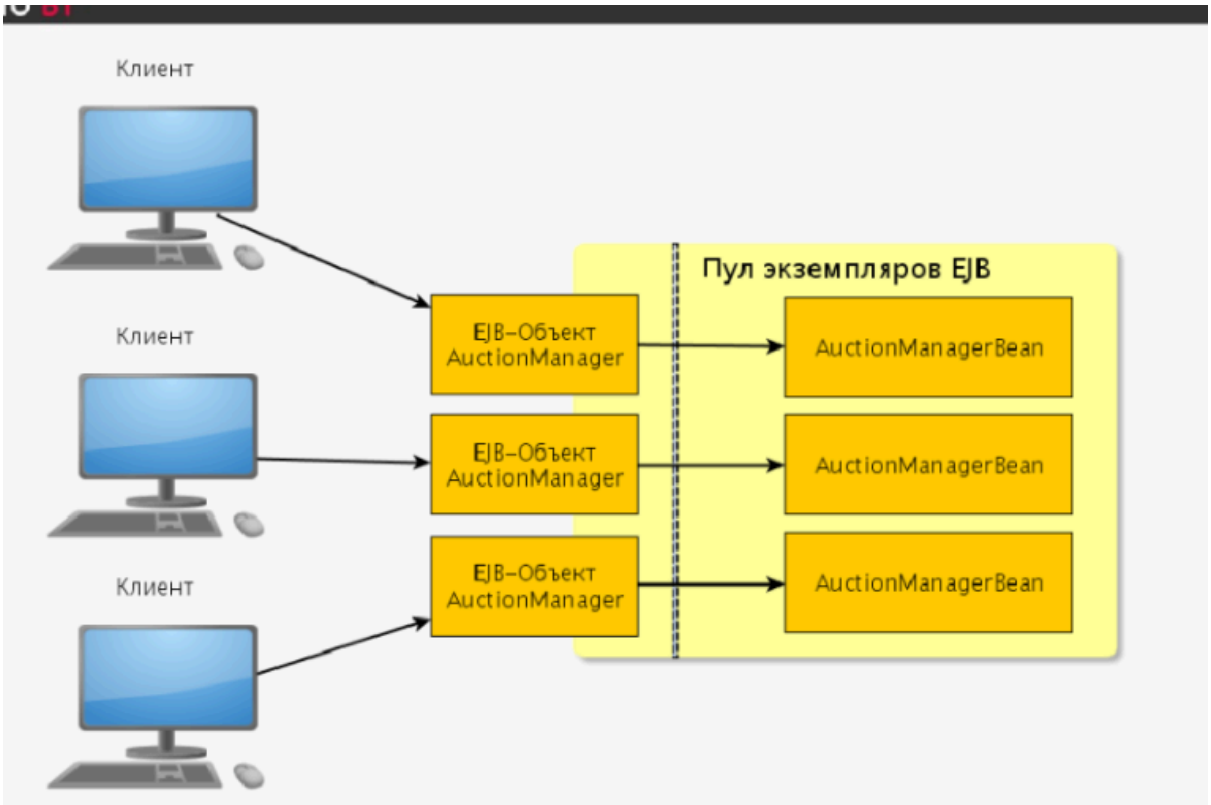
Stateful

- «Привязываются» к конкретному клиенту.
- Можно сохранять контекст в полях класса.
- Масштабируются хуже, чем @Stateless.
- Объявление — аннотация @Stateful.

Есть понятие сессии. Привязываются к конкретному клиенту. Когда клиент обращается к Statefull бину в первый раз, формируется отдельная EJB сессия. Существовать она будет в течение установленного времени после обращения клиента к этому бину, после чего бин будет освобожден и сможет быть переиспользован под другого клиента.

Так как привязка к конкретному клиенту гарантирована, то можно сохранять данные в полях бина. Бин привязан к клиенту.

Statefull бин нужен, когда есть последовательность взаимозависимых операций, совершаемых одним и тем же клиентом. В остальных случаях лучше использовать Stateless, так как у него масштабируемость лучше.



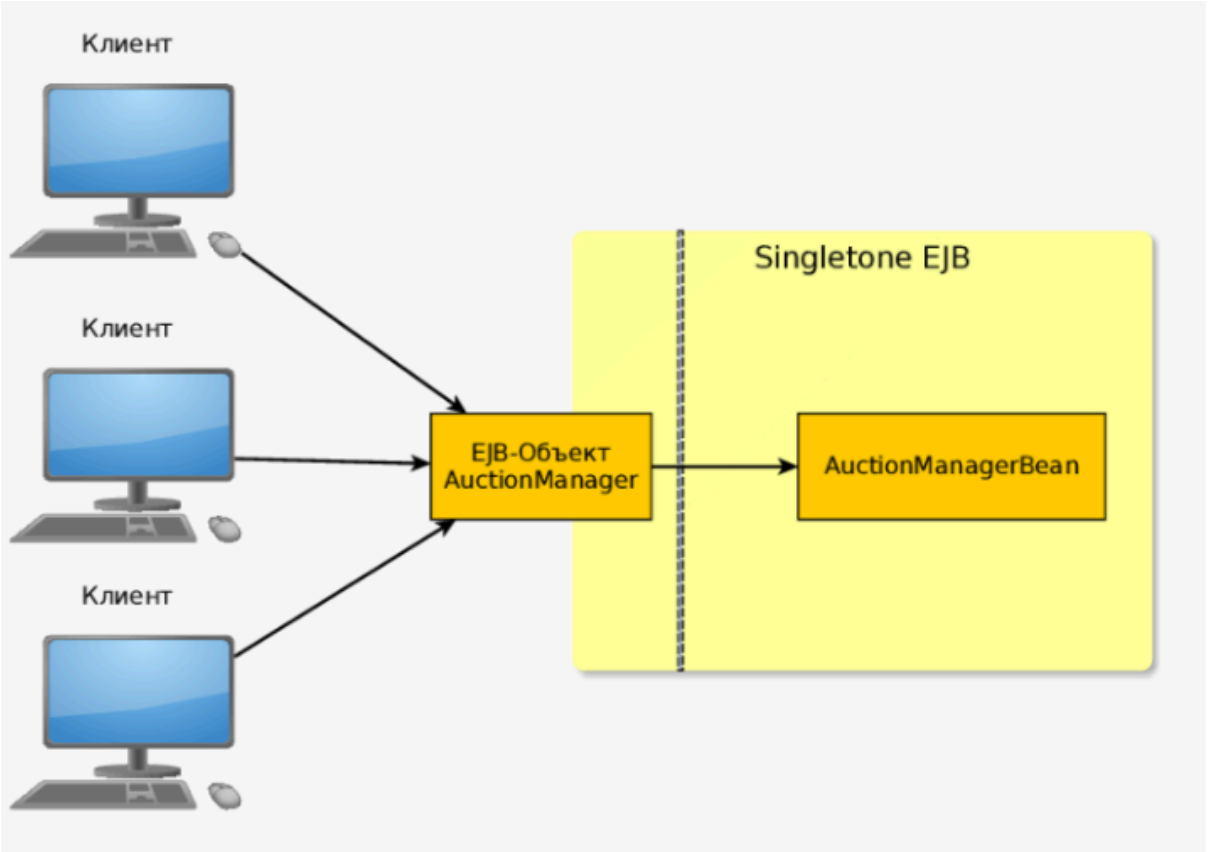
На каждый экземпляр EJB есть свой EJB-object. Привязка конкретного клиента к конкретному экземпляру EJB через привязку к EJB объекту.

Один EJB объект управляет только одним EJB бином.

Singleton

Контейнер гарантирует существование строго одного экземпляра такого бина.

Объявление — аннотация `@Singleton`.



Один EJB объект, который управляет одним EJB.

Все клиенты ходят на один EJB.

В плане масштабирования - это худший вариант, хотя обычно предполагается, что с бина будут что-то читать, что в плане масштабирования неважно.

EJB Lite & Full

Java EE Web Profile позволяет класть EJB Lite прямо в `war`ник (где лежит веб-приложение), а не в отдельный `jar`ник, как в случае с обычными EJB.

На слайде показано, что в `war`нике не будет работать.

Timer Service - позволяет сделать бин, а внутри него с помощью аннотаций сделать некое подобие крона. Крон - планировщик, например, позволяющий, запускать что-то в 5 часов ночи. Timer Service позволяет с помощью аннотаций задать время, в которое EJB`шный бин будет просыпаться. Пример и подробности по ссылке: https://otus.ru/nest/post/322/.

Существует 2 варианта деплоя:

- В Web Profile положить Lite бины прямо в `war`ник
- Для обычных EJB бинов создать отдельный `jar`ник. И jar и war положить на Full Profile.

`EAR`

Виды архивов Java`ских приложений:

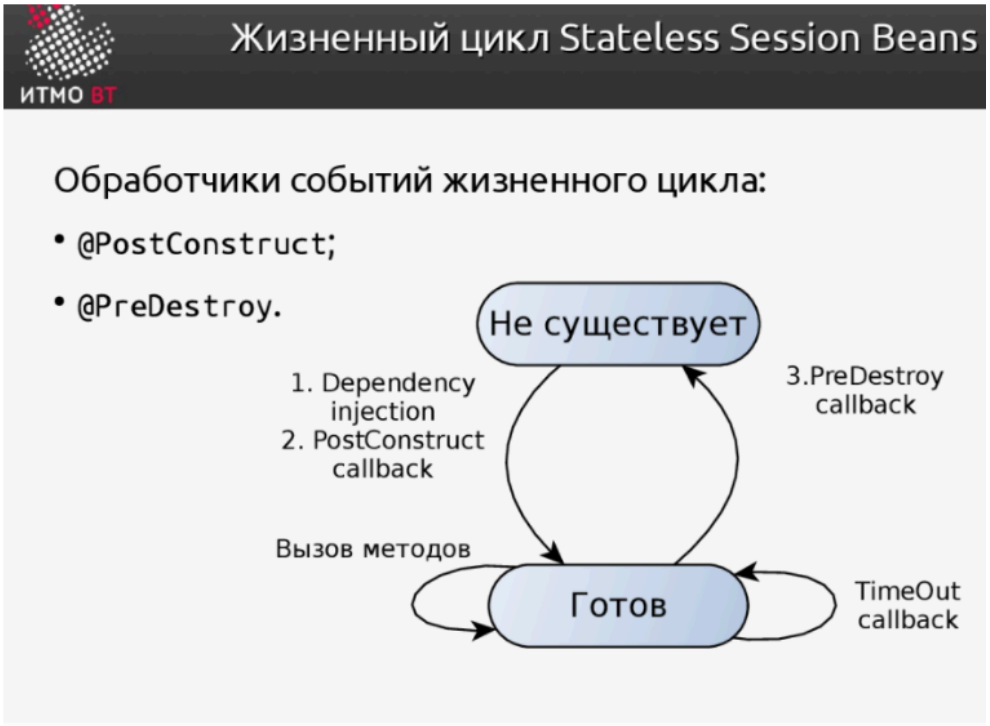
- `jar`
- `war`
- `rar`

  Resource Adapter. Позволяет внутрь Java EE приложения запихать адаптер на внешнюю систему (программа, написанная на Java с API).

- `ear`

  Enterprise Archive. Архив архивов. В него можно положить в определенной структуре разложенные архивы `jar`, `war` и `rar`. Работать с одним архивом удобнее., и развертывать понадобится только один архив, а не парочку.

Жизненный цикл Sessions Beans



▼ Конспект

Переход из "не существует" в "готов" происходит в тот момент, когда мы его инжектим.

▼ Конспект

У Stateful бина есть возможность активации/пассивации. То есть, когда бин не нужен, он переходит в пассивное состояние.

`@PostActivate callback` - обработчик события на переход их пассивного состояния в рабочее.

В остальном бин похож на Stateless.

Message-Driven Beans

MDB — это компоненты, логика которых является реакцией на события, происходящие в системе.

Особенности MDB:

- Реализуют интерфейс слушателя соответствующего типа сообщений (например, JMS).
- Нет возможности вызова методов «напрямую».
- Нет локальных и удалённых интерфейсов.
- Контейнер может создавать пулы MDB.
- При развёртывании MDB нужно регистрировать в качестве получателей сообщений.

С помощью MDB реализуется асинхронна обработка.

Типовой вариант: проснулся бин ночью и разослал новости, начислил проценты на счет.

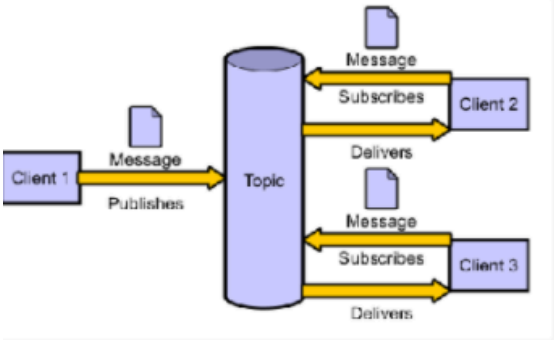
Message-Driven Beans предназначены для операций:

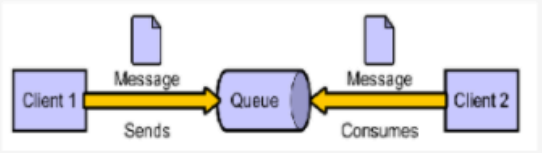
- Асинхронных
- Тяжелых

Механизмы доступа к MDB все те же, так как провайдеры топика и очереди зарегистрированы в JNDI.

JMS

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





JMS - API, которое описывает сервисы асинхронного обмена сообщениями.

Два вида:

- Подписка

  Клиент публикует сообщение, и его получают все подписчики.

- Очередь

  Клиент публикует сообщение, и его получает только тот, кто был первым в очереди. Тот, кто был первым в очереди, сообщение забирает, и оно их очереди пропадает.

Механика реализуется на общем Java EE`шном принципе. Java описывает спеки, поэтому провайдерами `topic` и `queue` являются независимые от Java EE решения.

Message Queue реализации:

- Active MQ
- Rabbit MQ

Возможности этих провайдеров существенно шире.

Получается следующее: Java EE содержит API для доступа к `topic` и `queue`. Дополнительно Application сервер должен содержать провайдера.

Пример кода Java Message Service'а есть по ссылке: java-online.ru/javax-jms.xhtml.

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, ...)

JTA

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

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

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

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

Транзакции могут быть объявлены:

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

Два способа управления транзакциями:

- Императивный (программный)

  Управляем транзакцией на уровне кода.

  BMT = Program - Bin-managed Transaction - трнзакции, управляемые программно. Мы сами управяем транзакциями.

  На слайде написано:

  `@TransactionManagment( BEAN )` - "BEAN" - мы управляем транзакциями программно.

  `@Resource UserTransaction ut` - заинжектили объект, который позволяет транзакциями управлять. Этот объект дает нам контейнер.

  Транзакции открываем и закрываем сами: `ut.begin()`, `ut.commit()`, `ut.rollback()`

- Декларативный (управляется контейнером)

  CMT = Declarative - Container-managed Transaction - транзакции, управляемые контейнером = Декларативное управление. Мы делегируем управление транзакциями на уровень контейнера.

  Контейнер на основании границ метода определяет моменты открытия и закрытия транзакции. И с помощью аннотаций мы можем подсказать контейнеру, как себя вести при входе в метод в рамках транзакции.

  Границами транзакции всегда является метод или набор методов. Контейнер, при заходе в метод, смотрит на аннотации и на их основании понимает, что с этим методом в рамках транзакции делать.

  ```
  @TransactionAttribute(TransactionAttributeType.REQUIRED)
  public void checkOut() {
    ...
  }
  ```

Спринг

Ioc и CDI

Бины имеют свои скопы(прототип, синглтон (+ сессия и запрос, если юзается в веб-приложении)) можно реализовать свои области видимости, бины помечаются аннотациями(@service, @component…) организация di идет с помощью @Autowired, @Inject, @Resource. можно инжектить с помощью конструктора, можно инжектнуть только какое то поле
IoC - это когда контейнер занимается порождением и управлением компонентов, и нам не приходится так часто видеть new в коде. Об управлении жизненным циклом компонентов в спринге написано тут.

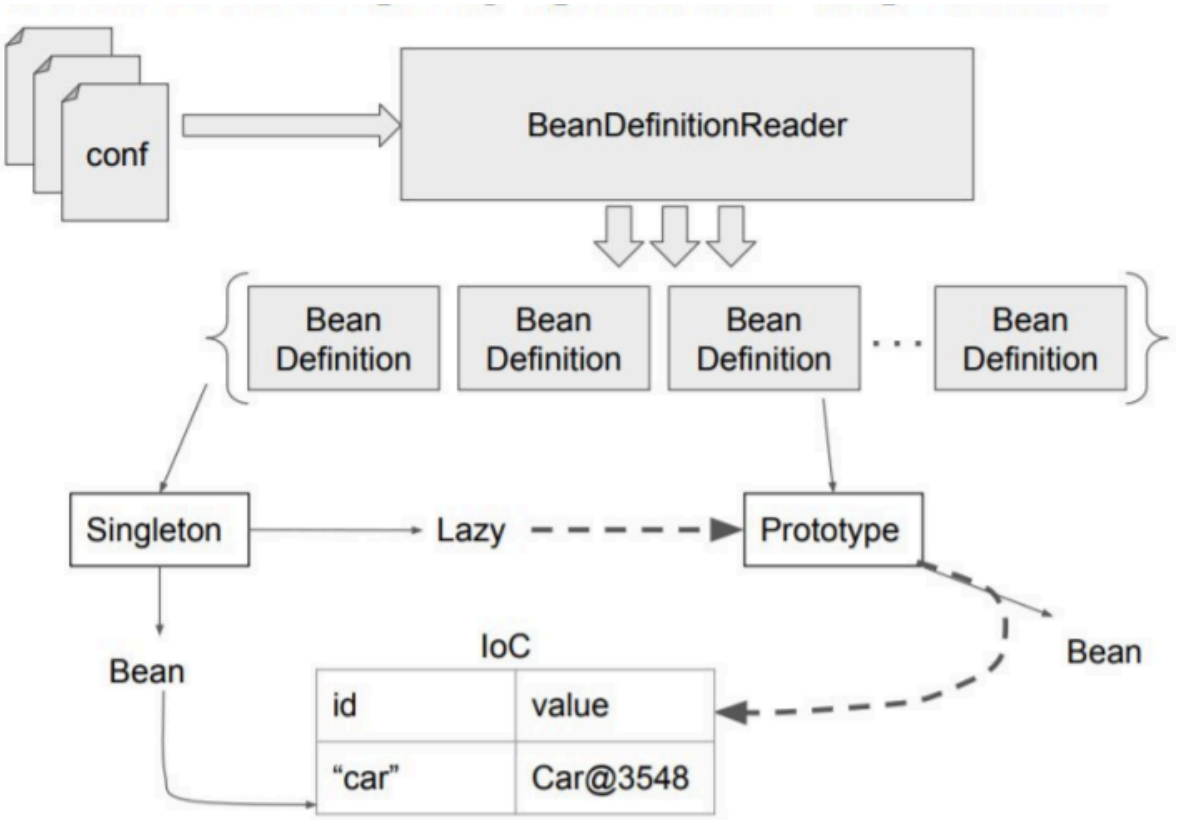
Spring отличие от java ee

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

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

Правда тут про спринг бут скорее. Но суть еще в том, что в java ee почти под каждый вид компонентов свой контейнер. Сам спринг по факту Ioc контейнер и есть куча спринговых модулей, которые мы можем добавить.

Жизненный цикл спринг приложения



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

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

Инициализация Spring Beans

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.

Rest в Spring

Есть контроллеры - специальные классы, помеченные аннотацией Controller или RestController (их отличие только в том, что в RestController на все метода автоматически навешивается аннотация ResponseBody, т.е что вернет метод, то засунется в тело ответа). Также стоит понимать, что Controller ничем не отличается от Componenta, просто программист отмечает, что этот класс используется как контроллер (отсылка к MVC). Методы помеченные аннотациями RequestMapping или GetMapping, PostMapping, etc (в них в основном передаются урлы (можем ничего не передавать), но есть еще всякие штуки по типу, что мы работаем с json или xml, которые можно указать в аннотации) становятся эндпоинтами. Есть еще аннотации, упрощающие жизнь при обработке запроса. Например, RequestBody в параметре метода сконвертирует тело запроса в этот самый параметр. Стоит упомянуть класс ResponseEntity - специальный класс для формирования ответа клиенту. Там мы можем задать ручками тело ответа, код ответа, хэдеры и прочее. По факту тут большая часть спринг мвц.

Виды DI в Spring

```
@Component
class Cake {

  private Flavor flavor;

  Cake(Flavor flavor) {
    Objects.requireNonNull(flavor);
    this.flavor = flavor;
  }
```

Через конструктор:

Просто передаем бин конструктору. Если несколько конструкторов,

то нужно указать Autowired к какому-то конструктору, чтобы Спринг знал, какой использовать для внедрения зависимостей.

Через сеттер:

```
@Component
class Cookie {

  private Topping toppings;

  @Autowired
  void setTopping(Topping toppings) {
    this.toppings = toppings;
  }
```

Спринг вызовет этот метод для внедрения зависимостей

Через поле:

```
@Component
class IceCream {

  @Autowired
  private Topping toppings;
```

Спринг просто внедрит зависимость в это поле. Считается плохой практикой.

Трехфазные конструкторы в Spring и Java EE

Мы знаем, что существует 2 фазы конструктора:

1) Обычный Java конструктор, в нем мы еще не проинициализированы зависимости в виде других бинов
2) Метод, помеченный аннотацией PostConstruct, там мы мы уже видим зависимости и можем производить нужные нам действия

Но в Спринг (хз насчет джава ее, может там тоже есть 3 фаза, но не хочу разбираться с этой помойкой, я бы сказал, что там нет такого, хотя использование аспектов частая тема) существует 3 фаза, когда мы можем завернуть наш бин в прокси объект и определить в нем поведение до вызова оригинального метода и после (аспекты). На моей практике это в основном используется для разделение бизнес логики, от каких-то вторичных действий, например логирования. Для этого нужно написать свою реализацию BeanPostProcessor’a и в методе invoke реализовать логику. Но опять же по моему опыту это не делают, а все такие действия реализуются аннотацией Aspect (думаю не стоит про нее говорить подробно, но в ее параметры мы передаем методы на которые нужно вызвать метод помеченный этой аннотацией)

Dependency lookup

В спринг есть 2 вида внедрения зависимостей DI и DL. Соответственно, при Dependency Injection у нас Ioc контейнер сам внедряет куда-то зависимость. Например,

```
@Component
public class MyClass{

  @Autowired
  MyBean myBean;

  // ...
}
```

Здесь мы сами не внедряем зависимость, а за нас это делает Spring.

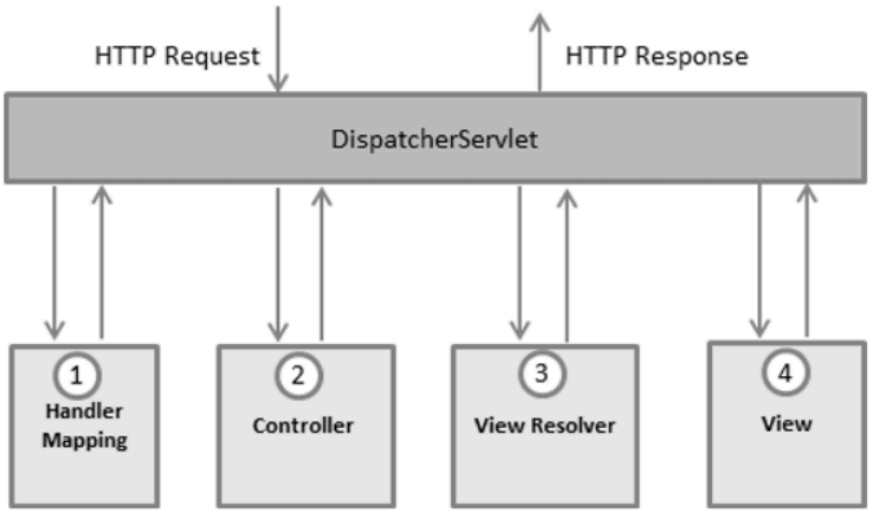
Dependency lookup же, это когда мы сами ищем зависимость для данного класса. Например, получим Application context и из него достанем бин.

```
ApplicationContext applicationContext = new ClassPathXmlApplicationContext("/application-context.xml");
MyBean bean = applicationContext.getBean("myBean");
```

Выглядит это не очень, и так почти никогда не делают.

Spring MVC

Многое расписано в Spring Rest. Спринг MVC основан на сервлетах. Существует диспетчер сервлет, который по маппингу делегирует обработку запросу какому-то методу контроллера. Реализует Model - View - Controller модель, где Модель отвечает за работу с данными, Controller - за обработку запроса, а View - за представление данных для пользователя.



- После получения HTTP-запроса DispatcherServlet обращается к интерфейсу HandlerMapping, который определяет, какой Контроллер должен быть вызван, после чего, отправляет запрос в нужный Контроллер.
- Контроллер принимает запрос и вызывает соответствующий служебный метод, основанный на GET или POST. Вызванный метод определяет данные Модели, основанные на определенной бизнес-логике и возвращает в DispatcherServlet имя Вида (View).
- При помощи интерфейса ViewResolver DispatcherServlet определяет, какой Вид нужно использовать на основании полученного имени.
- После того, как Вид (View) создан, DispatcherServlet отправляет данные Модели в виде атрибутов в Вид, который в конечном итоге отображается в браузере.

Конфигурируется стандартно в файле web.xml или при помощи аннотаций.

SPA

. SPA

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

RIA

Концепция RIA - подход к написанию интернет-приложений, предназначенных для выполнения задач, обычно выполняемых десктопными приложениями. Т.е это приложение, доступное через Интернет, богатое функциональностью традиционных десктопных приложений, не поддерживаемой браузерами непосредственно. Яркий пример тот же JSF, когда FacesServlet принимает http запрос и отдает http ответ, но внутри работает по событийной модели. Это удобно тем, что приложение пишется в десктопном стиле и на высоком уровне абстракции, а на значит меньше рутинной работы программисту. Но из этого

вытекают и минусы, что мы отходим от парадигмы написания веб-приложений полностью работающих по http, а высокий уровень абстракции усложняет работу “спуститься на уровень ниже”, на уровень протокола, что часто требуется. Из-за сложности архитектуры фреймворков сложнее сделать что-то, не предусмотренное разработчиками.

React

React js

React - это библиотека для разработки графических интерфейсов (причем не только веб-интерфейсов). React построен на компонентном подходе: существует корневой компонент App, вмонтированный в index.html, внутри которого вложены другие компоненты, образуя дерево компонентов. За отображение React интерфейса в DOM-дерево отвечает React DOM. При этом React реализует архитектуру Single Page Application - подход, при котором вместо множества сверстанных html-страниц, замапленных на разные URLы, есть JS-скрипт, который на лету управляет DOM-деревом - изменяет его, модифицирует, меняет вьюшки и так далее. Перед непосредственной отрисовкой (Render) React-приложение проходит стадию согласования (Reconciliation) строится новое дерево компонентов с измененным состоянием и сравнивается с предыдущим через хитрый, быстрый алгоритм. По итогу не требуется отрисовывать заново всё дерево, а только измененные его части

Структура в React. JSX

В реакте древовидная структура компонентов. Думаю здесь в целом можно повторить то, что написано сверху. jsx - синтаксический сахар для удобства верстки компонентов. Выглядит, как внедрение html кода в js. Позже это транслируется компилятором Babel в обычный React.createElement(...) (обычный способ создания элемента в реакте)

```
Т.е это const element = <h1>Hello, world!</h1>; транслируется в это const element = React.createElement("h1", null, "Hello, world!");
```

Также jsx поддерживает вставки js выражений через фигурные скобки.

React Router

Это апишка для навигации по реакт приложению. Для реализации такого API ReactRouter использует браузерное API, а именно объект window. ReactRouter слушает изменения состояния так называемого стэка истории (history stack).
Вся история навигации хранится в браузере в стеке, куда можно положить запись (push), удалить (pop) и заменить (replace).
ReactRouter используя обычные EventListener’ы отслеживает изменения в стеке и реагирует на них перерендерингом интерфейса, согласно маппингу. Маппинг просто связывает компонент и путь. Пример:

```
<BrowserRouter>
    <Switch>
        <Route path="/">
            <IndexPage/>
        </Route>
        <Route path="/main">
            <ApiMain/>
        </Route>
    </Switch>
</BrowserRouter>
```

State в redux и flux

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

Компоненты React. State & props. Умные и глупые компоненты

Компоненты в React позволяют декомпозировать интерфейс в отдельные независимые, реюзабельные части. По сути компоненты являются обычными JS-функциями, принимающими на вход параметры (props), которые обеспечивают кастомизацию компонентов, и возвращающими непосредственно описание интерфейса, который будет отрендерен на экране.

Задать компонент в React можно двумя способами:

1. Создать класс, расширяющий класс React.Component
2. Создать функцию, возвращающую view.

В основном различия между двумя подходами ограничиваются синтаксисом, но есть и довольные важные особенности, например, у функционального компонента отсутствует локальное состояние (т.к это обычная JS функция), и такой компонент использует состояние переданное извне (например через

props, useState(), менеджеров состояния). Также в компоненте-классе можно определить методы, управляющие жизненным циклом компонента (например componentDidMount() / componentWillMount()). В функциональном же компоненте приходится использовать альтернативу в виде хука useEffect().

В React принято разделение компонентов на умные и глупые.

Умные компоненты:

- управляют данными (модифицируют их, обрабатывают, возможно делают запросы на сторонние ресурсы)
- Используют API, состояние, библиотеки, методы управления жизненным циклом
- Упираются в функциональность компонента, а не в UI.

Глупые компоненты:

- Сконцентрированы на UI (кнопки, поля ввода и т. д.)
- Часто используют props для реюзабельности (например, в props можно передать title для кнопки)
- Обычно не используют сторонние библиотеки, кроме UI-ных.
- Могут иметь состояние, но супер локальное (например: поле для ввода имеет состояние непосредственно хранящее текущий ввод, но это не является состоянием приложения в целом)

Angular

Краткая выжимка

У нас есть Angular 2+, он же Angular CLI.
Общий принцип основан на том, что наше приложение состоит из модулей, который мы можем improt-ить между собой. В каждом модуле у нас есть компоненты, которые связаны с представлением (то есть с вьюшкой). Вообще, компонент - это класс, в котором есть методы, поля и всё такое. В компоненте мы реализуем всю логику того, как должна отрабатывать вьюшка. Компоненты можно вкладывать друг в друга.
Компонент и представление связаны двунаправленной связью (то есть ввели что-то в input на странице - поменялась какая-то переменная в компоненте и наоборот)
Также есть сервисы. Они реализуют ту логику, которая не меняют вьюшку (обмен по http, сохранение токена и тд).
Мы можем использовать компоненты и сервисы в других компонентах (и сервисах).
Для этого мы используем DI: помечаем нужный класс @Injectable (тот, который хотим где-то юзать) и вставляем его в конструктор того, где хотим юзать. Готово
Также у нас есть фильтры на странице (как и везде, фильтруют данные) и директивы (ngIf, ngFor , думаю понятно зачем они)
CSS мы добавляем через метапрограммирование (в декоратор добавляем либо

правила, либо готовые файлы)
Обмен с серваком у нас через HttpClientModule (его надо надо в главный модуль)

Angular. Отличие AngularJS и Angular CLI

ТМО ВТ

(Две большие разницы).

Angular -- открытая и свободная платформа от Google для разработки веб-приложений.

- Написана на языке TypeScript.
- Первый релиз (2.0) -- сентябрь 2016 г. Актуальная версия -- 7.0 (октябрь 2018 г.)
- Является развитием проекта AngularJS от той же команды (разрабатывается с 2009 г.)

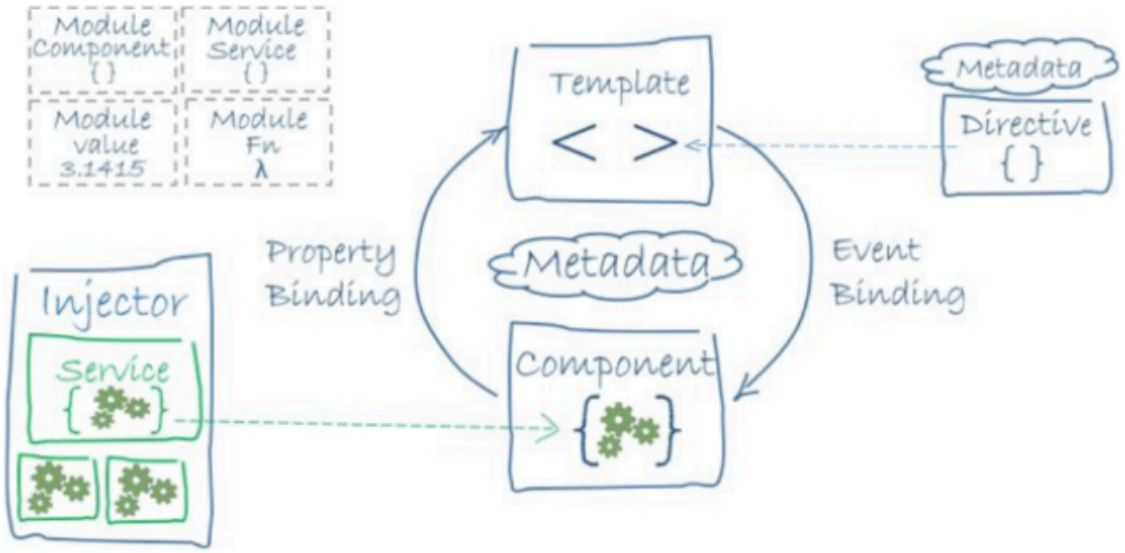
AngularJS относится ко всем версиям 1. x фреймворка. Он преобразует фиксированный HTML в динамический HTML. **Angular** — это обновление до **AngularJS**, которое работает быстрее, предлагает модульную структуру, использует интерфейс командной строки и, помимо других преимуществ, упрощает создание динамических SPA. 12 авг. 2020 г.

Angular. Архитектура и основные принципы разработки приложений

Angular - открытая и свободная платформа от Google для разработки веб-приложений. Написана на языке *Typescript*. Является развитием *AngularJS* от той же команды

Для разработки нужно настроить сборочное окружение на базе *NodeJS* и *npm*.

- Приложение состоит из *модулей* (*NgModules*)
- Модули обеспечивают контекст для *компонентов* (*components*)
- Из компонентов строятся *представления* (*views*)
- Компоненты взаимодействуют с *сервисами* (*services*) с помощью *DI*



Angular: модули, компоненты, сервисы и DI.

Модули (NgModules)

- Не совсем то же самое, что модули в ES6 (хотя и похожи).
- Каждый модуль обеспечивает *контекст компиляции* для одного или группы компонентов.
- Могут связывать компоненты с нужными для их работы сервисами.
- Группировка компонентов по модулям -- на усмотрение программиста.
- 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).
- Могут использоваться для реализации *загрузки по требованию* (*lazy loading*).

- Каждый модуль -- класс TS с декоратором `@NgModule()`.
- 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).

```
import { NgModule }      from '@angular/core';
import { BrowserModule } from '@angular/platform-browser';
@NgModule({
  imports:      [ BrowserModule ],
  providers:    [ Logger ],
  declarations: [ AppComponent ],
  exports:      [ AppComponent ],
  bootstrap:    [ AppComponent ]
})
export class AppModule { }
```

Компоненты

- У всех компонентов внутри модуля общий контекст компиляции.
- Компоненты и их шаблоны формируют представления (views).
- Компонент может содержать иерархию представлений (view hierarchy)
- Каждый компонент содержит корневое представление (host view)

- Каждый компонент - отдельный класс
- Компонент контролирует область экрана, называемую представлением (view)
- Angular управляет жизненным циклом компонентов

src/app/hero-list.component.ts (class)

```
export class HeroListComponent implements OnInit {
  heroes: Hero[];
  selectedHero: Hero;

  constructor(private service: HeroService) { }

  ngOnInit() {
    this.heroes = this.service.getHeroes();
  }

  selectHero(hero: Hero) { this.selectedHero = hero; }
}
```

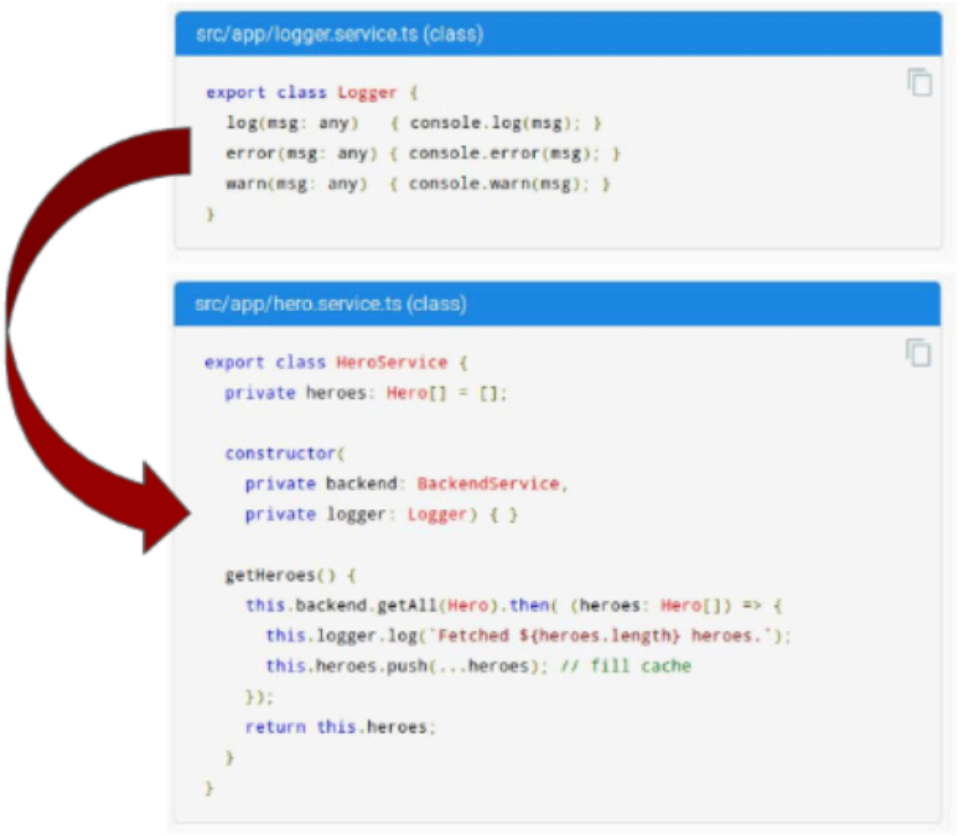
Метаданные компонента задаются с помощью декоратора @Component. Они сообщают рантайму о том, что это за компонент, и где искать его составляющие.

src/app/hero-list.component.ts (metadata)

```
@Component({
  selector:    'app-hero-list',
  templateUrl: './hero-list.component.html',
  providers:  [ HeroService ]
})
export class HeroListComponent implements OnInit {
/* . . . */
}
```

Сервисы и DI

- Сервисы (services) реализуют какие-либо действия, не формируя представление
- 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



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

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

Стандартные задачи сервисов:

- Предоставление данных приложению. Сервис может сам хранить данные в памяти, либо для получения данных может обращаться к какому-нибудь источнику данных, например, к серверу.
- Сервис может представлять канал взаимодействия между отдельными компонентами приложения
- Сервис может инкапсулировать бизнес-логику, различные вычислительные задачи, задачи по логгированию, которые лучше выносить из компонентов. Тем самым код компонентов будет сосредоточен непосредственно на работе с представлением. Кроме того, тем самым мы также можем решить проблему повторения кода, если нам потребуется выполнить одну и ту же задачу в разных компонентах и классах

- Компоненты могут использовать сервисы с помощью DI
- Для того, чтобы класс можно было использовать с помощью DI, он должен содержать декоратор @Injectable()

- 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

src/app/hero-list.component.ts (constructor)

```
constructor(private service: HeroService) { }
```

- Для каждого сервиса должен быть зарегистрирован как минимум один провайдер. Его можно задать в метаданных самого сервиса, в метаданных модуля или в метаданных компоненты

Для каждого сервиса должен быть зарегистрирован как минимум один провайдер.

Способы задания провайдера:

- в метаданных самого сервиса --->

```
@Injectable({
 providedIn: 'root',
})
```

- в метаданных модуля ---------->

```
@NgModule({
  providers: [
  BackendService,
  Logger
 ],
 ...
})
```

- в метаданных компонента -------->

```
@Component({
  selector:    'app-hero-list',
  templateUrl: './hero-list.component.html',
  providers:  [ HeroService ]
})
```

Angular. Шаблоны страниц, жизненный цикл компонентов, подключение CSS.

Шаблоны страниц

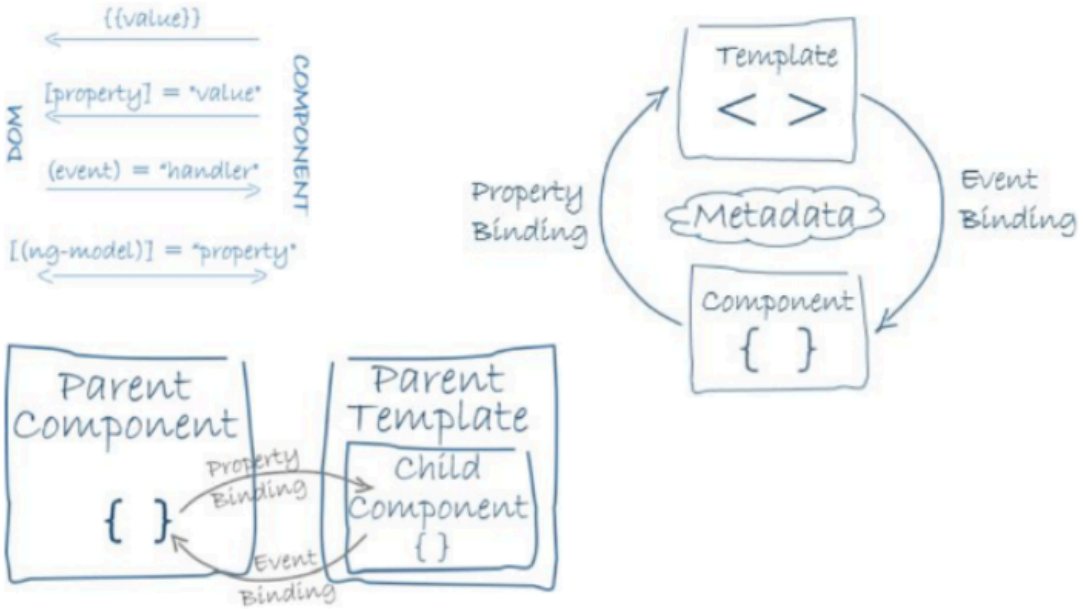
- Представление (view) компонента задаётся с помощью шаблона (template)
- Представления часто группируются иерархически
- Компонент может содержать иерархию представлений (view hierarchy), которая содержит встроенные представления (embedded views) из других компонентов

- Шаблон поход на обычный HTML
- Взаимодействие с классом компонента осуществляется с помощью ссылок на его свойства (data binding)
- Также можно использовать фильтры (pipes) и директивы (directives)

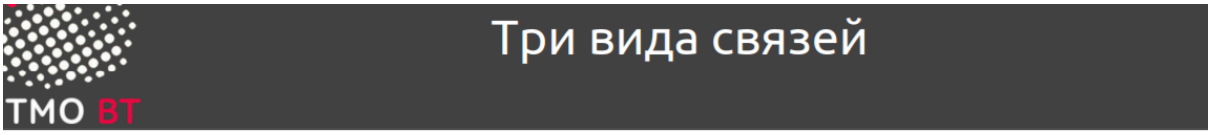
src/app/hero-list.component.html

```
<h2>Hero List</h2>

<p><i>Pick a hero from the list</i></p>
<ul>
  <li *ngFor="let hero of heroes" (click)="selectHero(hero)">
    {{hero.name}}
  </li>
</ul>

<app-hero-detail *ngIf="selectedHero" [hero]="selectedHero"></app-
hero-detail>
```

Связь между шаблоном и свойствами компонента - двунаправленная



Связи



src/app/hero-list.component.html (binding)

```
<li>{{hero.name}}</li>
<app-hero-detail [hero]="selectedHero"></app-hero-detail>
<li (click)="selectHero(hero)"></li>
```

- `{{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

- Позволяют осуществлять преобразование формата отображаемых данных (например, дат или денежных сумм) прямо в шаблоне
- Фильтры могут объединяться в последовательности (pipe chains)
- Фильтры могут принимать аргументы

```
<!-- Default format: output 'Jun 15, 2015'-->
 <p>Today is {{today | date}}</p>

<!-- fullDate format: output 'Monday, June 15, 2015'-->
<p>The date is {{today | date:'fullDate'}}</p>

 <!-- shortTime format: output '9:43 AM'-->
 <p>The time is {{today | date:'shortTime'}}</p>
```

Директивы

ITMO BT

Директивы (directives)

- Инструкции по преобразованию DOM.
- Создаются с помощью декоратора `@Directive()`.
- Все компоненты, технически -- директивы.
- Два вида:
  - Структурные директивы (structural directives):

src/app/hero-list.component.html (structural)

```
<li *ngFor="let hero of heroes"></li>
<app-hero-detail *ngIf="selectedHero"></app-hero-detail>
```

  - Директивы-атрибуты (attribute directives):

src/app/hero-detail.component.html (ngModel)

```
<input [(ngModel)]="hero.name">
```

Включение CSS

Для установки стилей в директиве @Component определено свойство **styles**:

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

@Component({
    selector: 'my-app',
    template: `<h1>Hello Angular 13</h1>
            <p>Angular 13 представляет модульную архитектуру приложения</p>`,
    styles: [`
            h1, h2{color:navy;}
            p{font-size:13px; font-family:Verdana;}
    `]
})
export class AppComponent { }
```

При использовании стилей следует учитывать, что они применяются локально только к разметке, управляемой компонентом. Например, если на странице будут элементы вне области управления компонентом, то к ним уже не будут применяться стили. Например:

```
<body>
    <my-app>Loading...</my-app>
    <h2>Подзаголовок</h2>
</body>
```

Если бы заголовок h2 здесь располагался бы в шаблоне компонента, то к нему применялся бы стиль. А так он не будет стилизован:

Селектор :host

Селектор **:host** ссылается на элемент, в котором хостится компонент. То есть в данном случае это элемент `<my-app></my-app>`. И селектор :host как раз позволяет применить стили к этому элементу:

```
styles: [`
    h1, h2{color:navy;}
    p{font-size:13px;}
    :host {
        font-family: Verdana;
        color: #555;
    }
`]
```

Подключение внешних файлов

Если стилей много, то код компонента может быть слишком раздут, и в этом случае их вынести в отдельный файл css. Так, создадим в одной папке с классом компонента (который по умолчанию располагается в папке app) новый файл **app.component.css** со следующим содержимым:

```
h1, h2{color:navy;}
p{font-size:13px;}
:host {
    font-family: Verdana;
    color: #555;
}
```

Затем изменим код компонента:

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

@Component({
    selector: 'my-app',
    template: `<h1>Hello Angular 13</h1>
            <p>Angular 13 представляет модульную архитектуру приложения</p>`,
    styleUrls: ['./app.component.css']
})
export class AppComponent { }
```

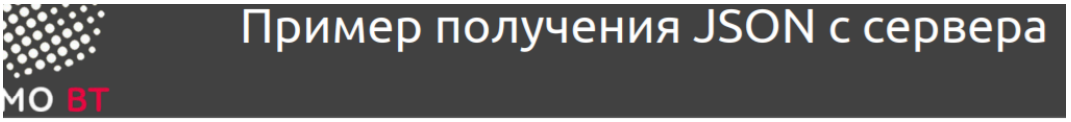
Параметр `styleUrls` позволяет указать набор файлов css, которые применяются для стилизации. В данном случае предполагается, что файл css располагается в проекте в папке app.

Angular. Клиент-серверное взаимодействие, создание, отправка и валидация данных форм.

- Взаимодействие реализуется с помощью сервиса *HttpClient*
- Для его использования необходимо импортировать модуль HttpClientModule в свой AppModule
- После импорта модуля можно заинжектить HttpClient

app/config/config.service.ts (excerpt)

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

@Injectable()
export class ConfigService {
  constructor(private http: HttpClient) { }
}
```



- Файл на сервере:

assets/config.json

```
{
  "heroesUrl": "api/heroes",
  "textfile": "assets/textfile.txt"
}
```

- Сервис:

app/config/config.service.ts (getConfig v.1)

```
configUrl = 'assets/config.json';

getConfig() {
  return this.http.get(this.configUrl);
}
```

Компонент:

app/config/config.component.ts (showConfig v.1)

```
showConfig() {
  this.configService.getConfig()
    .subscribe((data: Config) => this.config = {
        heroesUrl: data['heroesUrl'],
        textfile:  data['textfile']
    });
}
```

VueJS

Vue.js - это JS-фреймворк с открытым исходным кодом для разработки UI. Построен на архитектуре MVVM; может быть использован для разработки SPA в реактивном стиле - представление меняется по мере изменения модели.

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

Такое разделение позволяет ускорить разработку и поддерживаемость программы - можно менять один компонент, не затрагивая код другого.

ViewModel не может общаться с View напрямую. Вместо этого она представляет легко связываемые свойства в виде команд. View может привязываться к этим свойствам, чтобы получать информацию из ViewModel и вызывать на ней команды (методы). Это не требует того, чтобы View знала о ViewModel.

Экземпляр

- Любое приложение имеет, как минимум, один центральный экземпляр
- Для каждого файла HTML возможно любое количество экземпляров
- Экземпляр привязывается к узлу DOM с помощью свойства `el`:

```
var vm = new Vue({
  el: "body",
  data: {
    message: "Привет Мир!",
    items: [
      "это",
      "и",
      "есть",
      "Array/Массив"
    ]
  }
});
```

Опции экземпляра

$data

Объект с данными, за которыми осуществляет наблюдение экземпляр компонента. Экземпляр компонента проксирует доступ к свойствам объекта данных.

$props

Объект, содержащий текущие входные параметры, которые получил компонент. Экземпляр компонента проксирует доступ к свойствам объекта входных параметров.

$el

Корневой элемент DOM, которым управляет экземпляр компонента.

$options

Опции инициализации, используемые для текущего экземпляра компонента. Полезно, если потребуется добавить пользовательские свойства в опции

$parent

Родительский экземпляр, если таковой имеется

$root

Экземпляр корневого компонента текущего дерева компонентов. Если у текущего экземпляра нет родителя, то значением будет он сам

$slots

Используется для программного доступа к содержимому, распределяемому с помощью слотов. Каждый именованный слот имеет соответствующее свойства (например, содержимое `v-slot:foo` будет доступно через `this.$slots.foo()`). В свойстве `default` будут либо узлы, не попавшие в какой-либо именованный слот, либо содержимое `v-slot:default` .

Доступ к this.$slots пригодится при создании компонента с помощью render-функции.

```
<blog-post>
  <template v-slot:header>
    <h1>Обо мне</h1>
  </template>

  <template v-slot:default>
    <p>
      Какое-то содержимое страницы, которое будет доступно в $slots.default.
    </p>
  </template>

  <template v-slot:footer>
    <p>Copyright 2021 Evan You</p>
  </template>
</blog-post>
```

```
const app = createApp({})

app.component('blog-post', {
  render() {
    return h('div', [
      h('header', this.$slots.header()),
      h('main', this.$slots.default()),
      h('footer', this.$slots.footer())
    ])
  }
})
```

$methods

```
const app = Vue.createApp({
  data() {
    return { count: 4 }
  },
  methods: {
    increment() {
      // `this` will refer to the component instance
      this.count++
    }
  }
})
```

```
<button @click="increment">Up vote</button>
```

```
<span :title="toTitleDate(date)">
  {{ formatDate(date) }}
</span>
```

Компоненты

- Позволяют расширить функциональность экземпляров
- В отличие от экземпляров, не привязываются к узлам HTML, а содержат собственную разметку

```
// Определение компонента и глобальная регистрация
Vue.component('my-component', {
  template: '<div> это новый компонент < / div>'
})

// Создание экземпляра
new Vue({
  el: '#example'
})

<!-- HTML -->
<div id="example">
  <my-component></my-component>
</div>
```

Директивы

- Позволяют выполнять различные операции, например, итерацию по массиву или включение элементов в DOM по условию
- В разметке представляют собой атрибуты тегов

```
<div id="conditional-rendering">
  <span v-if="seen">Сейчас меня видно</span>
</div>

const ConditionalRendering = {
  data() {
    return { seen: true }
  }
}

Vue.createApp(ConditionalRendering)
   .mount('#conditional-rendering')
```

Жизненный цикл компонентов

Можно привязывать обработчики к фазам и событиям жизненного цикла компонентов

```
new Vue({
  // Вызывается, когда компонент будет видно,
  // например, через v-if vue или маршрутизатор.
  mounted: function () {
    console.log('Этот компонент был ' +
                'интегрирован в DOM ' + Date.now());
    this.$next(() => console.log('Теперь ' +
               'компонент готов к работе.'))
  },
  //Вызывается, когда компонент удаляется из DOM.
  destroyed: function() {
    removeListener(XY);
  }
})
```

Маршрутизация и навигация

Навигация происходит с помощью VueRouter

```
<div id="app">
  <router-view></router-view>
</div>

...
<script>
  ...
  const User = {
    template: '<div>User {{ $route.params.id }}</div>'
  };

  const router = new VueRouter({
    routes: [
      { path: '/user/:id', component: User }
    ]
  });
  ...
</script>
```

```
const NotFound = { template: '<h2>Page Not Found</h2>' }
const Home = { template: '<h2>Home Page</h2>' }
const Products = { template: '<h2>Products Page</h2>' }
const About = { template: '<h2>About Page</h2>' }

const routes = [
  { path: '/', component: Home },
  { path: '/products', component: Products },
  { path: '/about', component: About },
  { path: '*', component: NotFound }
];

const router = new VueRouter({
    mode: 'history',
    routes: routes
});

new Vue({
  el: '#app',
  router: router
})
```

```
<ul>
    <li><router-link to="/" exact>Home</router-link></li>
    <li><router-link to="/products">Products</router-link></li>
    <li><router-link to="/about">About</router-link></li>
</ul>
```

Экосистема

Официальные инструменты для разработчика

- Devtools - браузерный плагин для отладки приложений
- Vue CLI - консольные утилиты разработчика
- Vue Loader - загрузчик на базе webpack, позволяющий создавать компоненты Vue в формате Single-File Components (SFCs)

Официальные библиотеки

- Vue Router - библиотека для реализации маршрутизации и навигации в SPA
- Vuex - библиотека управления состоянием на базе Flux
- Vue Server Renderer - инструментарий для Server-Side Rendering

Практика

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

```java
@Singleton
public class SleepwalkerBean implements Sleepwalker {
    @PersistenceContext(unitName="somepersistence-unit")
    private EntityManager em;

    @Schedule(hour="0", minute="0", second="0")
    public void wakeUp() {
        em.createQuery("SELECT * FROM н_люди;")
        .getResultList()
        .forEach(el -> {
            Field[] fields = el.class.getFields();
            for (Field f : fields) {
                if (f.getAnnotation(Transient.class) != null)
                    continue;

                f.setAccessible(true);
                System.out.println(f.getName() + ": " + String.valueOf(f.get(el)));
            }
            System.out.println();
        });
    }
}
```

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

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

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

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

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

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

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

    HashMap results;

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

    }

}
```

Реализовать бин, который считает количество минут со старта приложения (или рестарта сервера)

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

Сервер JMS, который осуществляет рассылку спама всем подписчикам топика "lol"

```java
public class JmsTopicExample {
    public static void main(String[] args) throws URISyntaxException, Exception {
        BrokerService broker = BrokerFactory.createBroker(new URI(
                "broker:(tcp://localhost:61616)"));
        broker.start();
        Connection connection = null;
        try {
            ConnectionFactory connectionFactory = new ActiveMQConnectionFactory(
                    "tcp://localhost:61616");
            connection = connectionFactory.createConnection();
            Session session = connection.createSession(false,
                    Session.AUTO_ACKNOWLEDGE);
            Topic topic = session.createTopic("lol");
            connection.start();
            String payload = "Spam";
            Message msg = session.createTextMessage(payload);
            MessageProducer producer = session.createProducer(topic);
            producer.send(msg);
            session.close();
        } finally {
            if (connection != null) {
                connection.close();
            }
            broker.stop();
        }
    }
}
```

EJB калькулятор для 4-х операций

можно в целом подрубить lombok и сделать класс @Data

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

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

```
@ManagedBean(name=”myBean”)
@NoneScoped
class MyManagedBean {
        //тут у бина какие-то поля, методы и тд
}
```

по идее мы потом должны заинжектить данный бин в другой, у которой есть нормальный scope

```
@ManagedBean(name=”myOtherBean”)
@ApplicationScoped
public class Foo {
        @ManagedProperty(value = "#{myBean}")
        private MyManagedBean myManagedBean;
        public MyManagedBean getBar(){
                return this.myManagedBean;
        }
        public void setBar(MyManagedBean  t){
                this.myManagedBean = t;
        }
}
```

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

допустим кнопка генерит action main

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

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

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

Xml конфигурация Spring бина

```
<bean id="coin" class="ru.easyjava.spring.CoinImpl">
      <constructor-arg type="java.util.Random">
         <bean class="java.util.Random"/>
      </constructor-arg>
   </bean>
```

Таски на Angular

тк примеров нет, то вставлю куски из лабы и поясню их как-то

Создание компонента

```
import { Component, OnInit} from '@angular/core';
import {UserService} from "../service/user.service";
import {TokenService} from "../token/token.service";
//в декораторе пишем всякие данные про компонент.
Вместо templateUrl можно вставить просто html
Вместо styleUrls можно вставить прям css
@Component({
 selector: 'app-home',
 templateUrl: './home.component.html',
 styleUrls: ['./home.component.css']
})
export class HomeComponent implements OnInit {

 username: string = "";
 password: string = "";
 token: string="";
 //так собственно видим пример DI (вставляем что-то через
конструктор и потом можем юзать данные переменные)
 constructor(private userService: UserService) {
 }
//метод который загрузке компонента работает
 ngOnInit(): void {

 }

 log(){
//пример того самого DI (ниже будет код сервиса)
   this.userService.log({username: this.username,password:
this.password})
```

```
      .subscribe((res:any) => {//какой-то код}

      }, (error: any) => alert("Wrong password"));
    this.password = "";
    this.username = "";
  }

  }
```

Создание сервиса

смысл тут не важен

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

//вот главный прикол, чтобы мы могли юзать DI
@Injectable({
  providedIn: 'root' //и указываем провайдера
})
export class TokenService {

  constructor() { }

  signOut(): void {
    window.sessionStorage.clear();
  }
}
```

Форма ПСЖ

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

@Component({
    selector: 'my-app',
    styles: [],
    template: `<form #myForm="ngForm" >
                    <div>
                        <label>Имя</label>
                        <input name="name" [(ngModel)]="name" required />
                    </div>
                    <div>
                        <label>Фамилия</label>
                        <input name="surname" [(ngModel)]="surname" required />
                    </div>
                    <div>
                        <button (click)="go()">Access</button>
                    </div>
                </form>

})
export class AppComponent {

    name: string = "";
    surname: string = "";


    go(){
                //отправка формы(можно через сервис, можно напрямую через HttpClient
        }
}
```

Angular: как написали бы интерфейс, который проверял бы вошел ли пользователь и возвращал бы форму входа, при неавторизованности

Основное что надо: подключить Router и через него оформить навигацию, подключить HttpClient через которого отправить всё

```
import { Injectable } from '@angular/core';
import { HttpClient, HttpHeaders } from '@angular/common/http';
import {Router} from '@angular/router';


@Injectable({
  providedIn: 'root'
})
export class TokenService {

  constructor(private http: HttpClient, private router: Router) { }
  signIn(data){
        this.http.post("someUrl", data).subscribe((res:any) => {
                if (res.token == ""){
                        this.router.navigate(['/main]);
```

```
                }else{
                        alert(“Success”);
                }
};
}
```

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

```
import { StrictMode } from "react";
import ReactDOM from "react-dom";
import React from "react";

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

function getList(str){
fetch('https://api.npms.io/v2/search?q='+str) .then(response =>
response.json()) .then(data => setState({
list = data;
}));


}

render() {
return(
  <div>
    <input onchange=”getList()” />
          <CompletionBox list=”this.state.list”/>
  </div>)
}
}


const rootElement = document.getElementById("root");
```

```
ReactDOM.render(
  <StrictMode>
    <Autocomplete />
  </StrictMode>,
  rootElement
);
```

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

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


class LoginForm extends React.Component {

    constructor(props) {
        super(props);
        this.state = {
            username: "",
            isAuth: false
        }
        this.auth = this.auth.bind(this);
    }

    auth() {
        let login = document.getElementById("login").value;
        let password = document.getElementById("password").value;
        axios.post("/login", {
            login, password
        }).then(() => {
            this.setState({
                username: login,
                isAuth: true
            });
        }).catch(error => {
            alert(error);
        })
    }

    render() {
        return (
            <div>
                <input id="login" name="login"/>
                <input id="password" name="password"/>
                <button type="submit" onClick={this.auth}/>
            </div>
```

```
    )
  }
}
```

CRUD на Spring Web MVC

```
@RestController
class EmployeeController {
  private final EmployeeRepository repository;

  EmployeeController(EmployeeRepository repository) {
    this.repository = repository;
  }

  @GetMapping("/employees")
  List<Employee> all() {
    return repository.findAll();
  }

  @PostMapping("/employees")
  Employee newEmployee(@RequestBody Employee newEmployee) {
    return repository.save(newEmployee);
  }

  @GetMapping("/employees/{id}")
  Employee one(@PathVariable Long id) {
    return repository.findById(id)
      .orElseThrow(() -> new EmployeeNotFoundException(id));
  }

  @PutMapping("/employees/{id}")
  Employee replaceEmployee(@RequestBody Employee newEmployee,
@PathVariable Long id) {
    return repository.findById(id)
      .map(employee -> {
        employee.setName(newEmployee.getName());
        employee.setRole(newEmployee.getRole());
        return repository.save(employee);
      })
      .orElseGet(() -> {
        newEmployee.setId(id);
        return repository.save(newEmployee);
      });
  }

  @DeleteMapping("/employees/{id}")
  void deleteEmployee(@PathVariable Long id) {
    repository.deleteById(id);
```

```
  }
}
```

Создание JSF-компонента

```
@FacesComponent(createTag = true, tagName = "helloComponent", namespace =
"http://example.com/tags")
public class HelloComponent extends UIComponentBase {
  @Override
  public String getFamily() {
      return "Greeting";
  }

  @Override
  public void encodeBegin(FacesContext context) throws IOException {
      String message = (String) getAttributes().get("message");
      LocalDateTime time = (LocalDateTime) getAttributes().get("time");
      String formattedTime = time.format(
              DateTimeFormatter.ofLocalizedDateTime(FormatStyle.MEDIUM));

      ResponseWriter writer = context.getResponseWriter();
      writer.startElement("p", this);
      writer.write("Message: " + message);
      writer.endElement("p");

      writer.startElement("p", this);
      writer.write("Time: " + formattedTime);
      writer.endElement("p");
  }
}

<t:helloComponent message="#{helloBean.message}"
time="#{helloBean.time}"/>
```

Создание формы на React + API-запрос

Создание формы на Angular + API-запрос

Создание формы на VueJS + API-запрос

Изменение стейта в React+Redux// сказали, редакса не будет

Изменение стейта в React+Flux

ActionTypes.js

```
const ActionTypes = { ADD_ITEM: "ADD_ITEM", REMOVE_ITEM: "REMOVE_ITEM"};
export default ActionTypes;
```

PhonesDispatcher.js

```
import {Dispatcher} from "flux";
export default new Dispatcher();
```

Actions.js

```
import ActionTypes from "./ActionTypes.js";
import PhonesDispatcher from "./PhonesDispatcher.js";

const Actions = {
  addItem(text) {
    PhonesDispatcher.dispatch({
      type: ActionTypes.ADD_ITEM,
      text,
    });
  },
  removeItem(text) {
    PhonesDispatcher.dispatch({
      type: ActionTypes.REMOVE_ITEM,
      text,
    });
  }
};

export default Actions;
```

Phones.js

```
import Immutable from "immutable";
import {ReduceStore} from "flux/utils";
import Actions from "./Actions.js";
import ActionTypes from "./ActionTypes.js";
import PhonesDispatcher from "./PhonesDispatcher.js";

class PhonesStore extends ReduceStore{
    constructor()
    {
        super(PhonesDispatcher);
    }
    getInitialState() {
        return Immutable.List.of("Apple iPhone 12 Pro", "Google Pixel 5");
    }
```

```
    reduce(state, action) {
        switch (action.type) {
            case ActionTypes.ADD_ITEM:
                if (action.text) {
                  return state.push(action.text);
                }
                return state;
            case ActionTypes.REMOVE_ITEM:
                let index = state.indexOf(action.text);
                if (index > -1) {
                    return state.delete(index);
                }
                return state;
            default:
                return state;
        }
    }
}
export default new PhonesStore();
```

AppView.js

```
import React from "react";

class AppView extends React.Component{

    constructor(props){
        super(props);
        this.state = {newItem: ""};

        this.onInputChange = this.onInputChange.bind(this);
        this.onClick = this.onClick.bind(this);
    }
    onInputChange(e){
        this.setState({newItem:e.target.value});
    }
    onClick(e){
        if(this.state.newItem){
            this.props.onAddItem(this.state.newItem);
            this.setState({newItem:" "});
        }
    }
    render(){

        let remove = this.props.onRemoveItem;
        return <div>
```

```
                <input type="text" value={this.state.newItem}
onChange={this.onInputChange} />
                <button onClick={this.onClick}>Добавить</button>
                <h2>Список смартфонов</h2>
                <div>
                    {
                        this.props.phones.map(function(item){

                            return <Phone key={item} text={item} onRemove={remove}
/>
                        })
                    }
                </div>
            </div>;
    }
}

class Phone extends React.Component{

    constructor(props){
        super(props);
        this.state = {text: props.text};
        this.onClick = this.onClick.bind(this);
    }
    onClick(e){
        this.props.onRemove(this.state.text);
    }
    render(){
        return <div>
                <p>
                    <b>{this.state.text}</b><br />
                    <button onClick={this.onClick}>Удалить</button>
                </p>
            </div>;
    }
}
export default AppView;
```

AppContainer.js

```
import AppView from "../views/AppView.js";
import {Container} from "flux/utils";
import React from "react";
import PhoneStore from "../data/PhoneStore.js";
import Actions from "../data/Actions.js";

class AppContainer extends React.Component
```

```
{
    static getStores() {
        return [PhoneStore];
    }
    static calculateState(prevState) {
        return {
            phones: PhoneStore.getState(),
            onAddItem: Actions.addItem,
            onRemoveItem: Actions.removeItem
        };
    }
    render() {
        return <AppView phones={this.state.phones}
                        onRemoveItem={this.state.onRemoveItem}
                        onAddItem={this.state.onAddItem}  />;
    }
}
export default Container.create(AppContainer);
```

EJB с арифметическими операциями

ManagedBean с арифметическими операциями

React компонент для хранения информации о банковской карте

```
export class CardComponent extends Component {

      constructor(props) {
            this.state = {
                  number=null,
                  date=null,
                  cvv=null,
                  name=null
            }
      }

      handleChange(e) {
            const val = e.target.value
            const name = e.target.name;
            //валидация
            this.state.name = value
      }

      function render() {
```

```
            <div>
                  <span>Number</span>
                  <input type={"text"} name={"number"}
keyFilter={/[1-9^\s]/} maxLength={"16"} minLength={"16"}
value={this.state.number} onChange={handleChange(this)}>
                  <span>Owner name</span>
                  <input type={"text"} name={"name"}
value={this.state.name} keyFilter={/[A-z^\s]/}
onChange={handleChange(this)}>
                  <span>Date</span>
                  <input type={"month"} name={"date"}
value={this.state.date} onChange={handleChange(this)}>
                  <span>CVV</span>
                  <input type={"text"} name={"cvv"} value={this.state.cvv}
minLength={"3"} maxLength={"3"} onChange={handleChange(this)}>
            </div>
      }
}
```