JAVA EE

Java Enterprise Edition (ранее Java2 Enterprise Edition/J2EE, позже Jakarta EE) - набор спецификаций и документаций, используется для корпоративной разработки.

Общие понятия, общая архитектура

Приложения строятся на основе компонентов, жизненным циклом которых управляют контейнеры.

Принципы IoC, CDI, Location Transparency

Inversion of Control - принцип согласно которому компонент системы по возможности не должен полагаться на детали реализации другого её конкретного компонента, есть и другая формулировка (в контексте Java EE), согласно которой управление жизненным циклом программы и взаимодействием компонентов перекладывается с программиста на контейнер.

В Java EE **IoC** реализуется с помощью Contexts and Dependency Injection - спецификации Java EE, позволяющая делать инъекции (автоматически внедрять связи между компонентами), управлять их жизненным циклом и контекстами

IoC – Инверсия управления – принцип программирования, каждый компонент системы должен быть как можно более изолированным от других.

Location Transparency - принцип, скрывающий физическое расположение данных от клиента, в Java EE реализован к примеру с помощью JNDI (Java Naming and Directory Interface) - абстрагирующего доступ к ассетам

(возможность удаленно работать с компонентами так, как будто они лежат рядом)

CDI – Контекст и инъекция зависимостей - Стандартная реализация IoC в Java EE:

- **Инъекцию зависимостей:** Внедрение зависимостей в управляемые бины.
- **Интерцепторы:** Перехват вызовов методов управляемых бинов.
- **Продусеры:** Создание объектов, управляемых контейнером.
- **Контексты:** Управление жизненным циклом и областью видимости бинов (например, запрос, сессия, приложение).

Профили платформы

Профили - наборы спецификаций Java EE, каждый из которых предоставляет возможности для разработки определённого типа продуктов. Full Profile содержит все возможные спецификации, а Web Profile - только необходимые для разработки веб-приложений.

• Полный профиль подходит для крупных корпоративных приложений с высокой нагрузкой, где требуется использование всех возможностей Java EE.

• Веб-профиль лучше подходит для легковесных веб-приложений, где не требуется сложная бизнес-логика или распределенные транзакции.

• MicroProfile идеально подходит для разработки микросервисов и облачных приложений, обеспечивая гибкость и легкость интеграции с другими сервисами.

Java EE: общие понятия, общая архитектура

Приложения строятся на основе компонентов, жизненным циклом которых управляют контейнеры.

Java EE – это платформа для разработки и развёртывания корпоративных приложений, предоставляет набор спецификаций (API): Servlet, JSP, JPA, EJB, JTA, JMS и другие.

JSF

JSF: Базовые понятия, характеристика технологии, реализуемая модель и прочие общие моменты

JSF (JavaServer Faces) — это фреймворк для разработки веб-приложений на Java, использующий компонентную модель для построения пользовательского интерфейса. Он предоставляет набор готовых компонентов, упрощающих создание сложных и интерактивных веб-страниц.
Faces Servlet, Faces Context, Managed Beans, EL, Tag Libraries
Реализует паттерн MVC
Поддерживает валидации, JSTL

JSF: Managed beans

Managed Beans — это классы Java, которые управляются контейнером JSF. Они используются для обработки данных и логики приложения. @ManagedBean, область видимости

JSF: Дерево компонентов и в принципе UI составляющая

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.

JSF: Конфигурация

Возможности: управление бинами, навигация, фильтры и слушатели, конверторы и валидаторы, кастомные компоненты.
JSF можно настраивать через файл конфигурации faces-config.xml, который обычно находится в папке WEB-INF. В этом файле можно определить Managed Beans, навигацию и другие параметры.

JSF: Жизненный цикл

- Восстановление представления (Restore View) : для запрошенной страницы либо извлекается дерево компонентов, если страница уже открывалась, либо создается новое дерево компонентов, если страница запрашивается впервые. Для компонентов запрашиваемой страницы восстанавливаются их прошлые состояния (форма заполняет вводимыми значениями).
- Применение значений запроса (Apply Request Values) : анализ HTTP запроса, объектам дерева компонентов присваиваются соответствующие им значения из запроса. Если к компоненту подключен конвертер, то значение обрабатывается/конвертируется. При возникновении ошибки, все последующие шаги пропускаются. Если компонент ввода (UIInput) содержит валидатор и имеет свойство immediate="true", то этот компонент будет валидироваться в этой фазе. Также, при необходимости, события (нажатие кнопки или ссылки) добавляются в очередь событий.
- Выполнение проверок (Process Validations) : преобразование строковых значений в "локальные значения" и применение валидации дерева компонентов. Если в результате валидации компонента возникает ошибка, то она сохраняется и JSF пропускает все последующие шаги обработки запроса до фазы "Формирования ответа" для предупреждения пользователя об ошибке.
- Обновление значений модели (Update Model Values) : обновление свойства managed bean информацией из дерева компонентов.
- Выполнение приложения (Invoke Application) : JSF обрабатывает события, которые были сгенерированы нажатием на кнопки и ссылки. На данном этапе также решаются вопросы, связанные с навигацией приложения, если это необходимо. Если один из компонентов формы имеет свойство immediate="true", то он должен был быть обработан в фазе "Применение значений запроса".
- Формирование ответа (Render Response) : JSF создает ответ, основываясь на данных, полученных на предыдущих шагах. Информация страницы обновляется данными из managed bean и генерируется html страница с помощью Renderers. **Если на предыдущих шагах происходили какие-либо ошибки, то они инкапсулируются в тег <messages>.**

JSF: Обработка событий

JSF поддерживает обработку событий через слушатели (event listeners). Например, вы можете обрабатывать события нажатия кнопки с помощью метода в Managed Bean.

<h:commandButton value="Submit" action="#{myBean.process}" />

JSF: Конверторы и валидаторы

Конверторы – преобразование данных (аннотация @FacesConverter("dateConverter"))

Валидаторы – проверяют корректность данных, есть встроенные (аннотация @FacesValidator("myValidator"))

CDI beans

CDI beans: общее понятие

CDI beans — это управляемые объекты, которые могут быть созданы, инъектированы и управляемы контейнером CDI. Они позволяют разработчикам легко управлять зависимостями и жизненным циклом объектов в приложении. (Контейнер управляет жизненным циклом, инъекция зависимостей, контексты)

CDI beans: аннотации

CDI предоставляет аннотации для работы и управления бинами

1. @Inject: Используется для инъекции зависимостей в поля, конструкторы или методы.

2. @Named: Позволяет назначить bean имя, что делает его доступным для использования в EL (Expression Language) в JSP или Facelets.

3. @ApplicationScoped: Определяет bean с жизненным циклом на уровне приложения. Один экземпляр будет создан и использоваться во всем приложении.

4. @SessionScoped: Определяет bean с жизненным циклом на уровне сессии. Экземпляр будет создан для каждой пользовательской сессии.

5. @RequestScoped: Определяет bean с жизненным циклом на уровне запроса. Экземпляр создается для каждого HTTP-запроса.

6. @Dependent: Указывает, что жизненный цикл bean зависит от жизненного цикла объекта, который его инъектирует.

7. @Produces: Используется для определения метода или поля, который создает bean и делает его доступным для инъекции.

8. @Qualifier: Позволяет создавать пользовательские аннотации для уточнения, какой именно bean следует инъектировать, если существует несколько вариантов

CDI beans: именование, "фабрики", "перехватчики"

Фабрика - **бин**, **создающий экземляры других бинов** при этом инъектировать можно любой класс, даже не бин, используя аннотации:

- @**Produces** - накидывается на метод-фабрику для создания бинов
- @**Disposes** (optional) - накидывается на метод вызываемый перед удалением создаваемого бина

Перехватчик - **бин, который слушает события жизненного цикла** бинов, тип которых задаётся аннотацией:

- @**AroundInvoke** - при вызове метода
- @**PostConstruct** - после конструктора

@**PreDestroy** - перед уничтожением

ORM и иерархия технологий работы с БД

ORM – технология, позволяющая работать с базами данных, используя ООП. (автоматический перевод данных). Сущности, сессии, запросы, связи.
СУБД – ORM – JPA – Библиотека – Код

ORM (Object to Relational Mapping) решает задачу преобразования данных из реляционной формы в объектную и наоборот, реализуется с помощью:

- JDBC
- ORM-фреймворки (Hibernate, ...)
- JPA 2.0

Иерархия устроена следующим образом:

1. Логика работы с данными
2. **JPA** (**J**ava **P**ersistance **A**PI) - спецификация API для управлениями объектами в БД
3. **ORM-фреймворк** - реализация JPA, берет на себя преобразование из объектов в таблицы и обратно
4. **JDBC** (**J**ava **D**ata**B**ase **C**onnectivity) - низкоуровневый API для работы с БД
5. **JDBC-драйвер** реализует взаимодействие с конкретной базой данных
6. База данных

JPA

JPA: общие принципы

1) Сущности – Java классы, соответствующие таблицам, каждая имеет уникальный ключ
2) Контекст персистенции – среда для управления сущностью (Он отслеживает состояние объектов и синхронизирует их с базой данных)
3) Менеджер сущностей - это интерфейс, который предоставляет методы для работы с сущностями, включая создание, чтение, обновление и удаление объектов. Он также управляет транзакциями.
4) Запросы
5) Транзакции

JPA: конфигурация

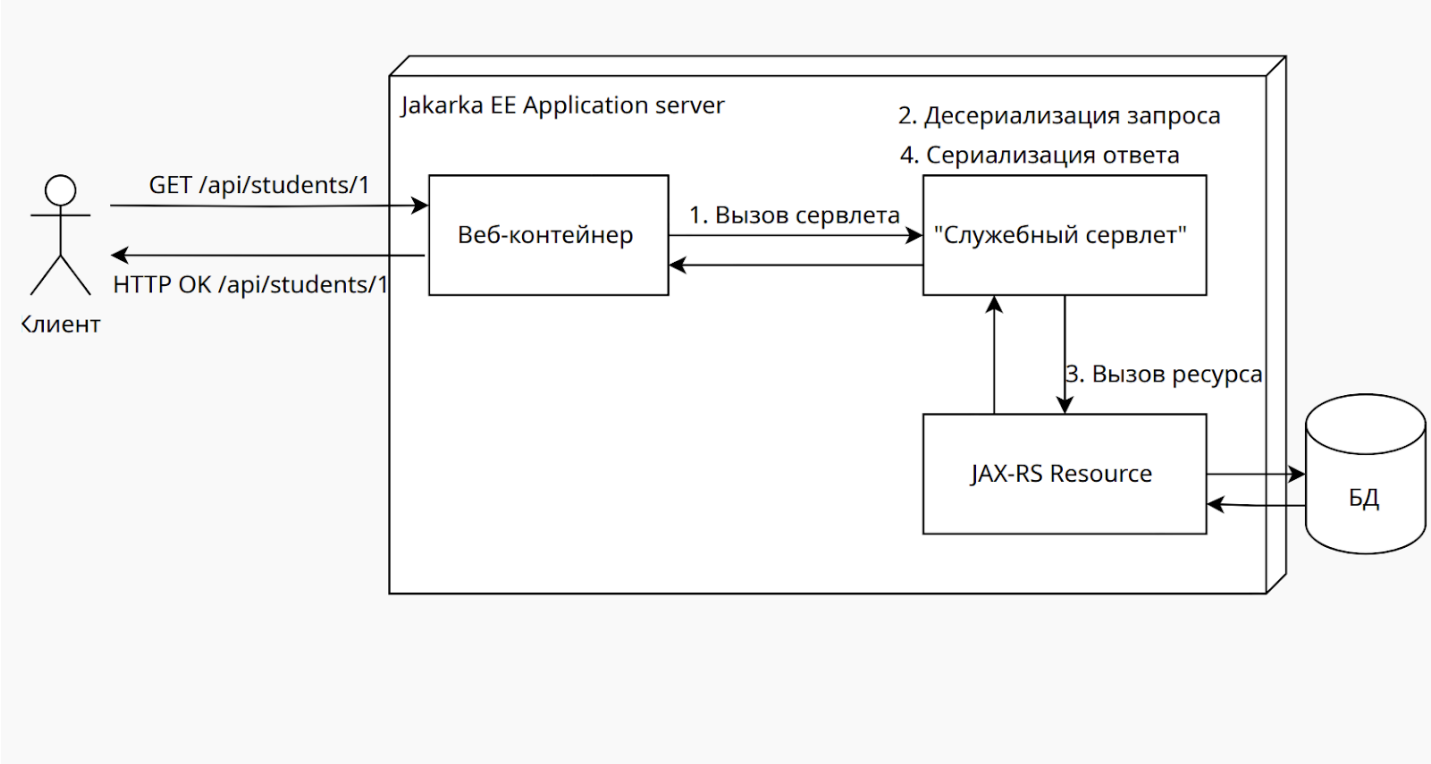
конфигурация включает в себя все необходимые данны для подключения к бд.
1) файл persistence.xml в META-INF
2) программно, создав экземпляр EntityManagerFactory

JPA: аннотации

JPA предоставляет несколько аннотаций для определения сущностей и их свойств.
1) @Entity
2) @Table
3) @Id
4) @Column
5) @OneToMany....
6) @Transient – поле не будет включено в бд
7) @Embeddable и @Embedded для сохранения встроенных объектов

JAX-RS: общие принципы

JAX-RS - это спецификация для создания RESTful веб-сервисов в Java, используются стандартные HTTP-методы, Stateless, Клиент-серверная архитектура.
Особенности: REST, Аннотации (Path, (методы: GET, POST, DELETE,PATH), @Produces и @Consumes – поддерживаемые типы данных , например json или XML).



JAX-RS: аннотации

@Path – путь до ресурса (класс или метод)
@GET, @POST, @PUT, @DELETE – HHTP-метод для метода
@PathParam, @QueryParam, @HeaderParam, @CookieParam
@Produces и @Consumes – поддерживаемые типы данных для приема и отправки
@Context - позволяет инъектировать контекстные объекты

ФРОНТ

Frontend: рендеринг

Рендеринг — это процесс преобразования кода (HTML, CSS, JavaScript) в визуальное представление на экране пользователя:
1) Клиентский рендеринг – генерация в браузере
2) Серверный рендеринг – генерация на сервере
3) Гибридный

Frontend: принципы MPA, SPA, PWA, CSR, SSG, SSR

Паттерны рендеринга:

- **CSR (Client Side Rendering) -** рендеринг на стороне клиента, контент рендерится JS-ом по сырым данным полученным с сервера
  Плюсы: меньше трафика
  Минусы: SEO, долгая первая отрисовка
- **SSR (Server Side Rendering) -** рендеринг на стороне сервера
  Плюсы: низкие требования к производительности на клиенте, высокая совместимость, SEO
  Минусы: больше трафика, больше серверных ресурсов
- **SSG (Server Side Generation) -** совмещает два предыдущих подхода, сервер отдаёт готовую страничку, которая при этом имеет возможность самостоятельно обновляться
  Плюсы: +-SEO, быстрая первая загрузка
  Минусы: сложно в поддержке и разработке

Устройства приложений:

- **MPA (Multiple Pages Application) -** устройство приложения, при котором ресурсы статичны и каждому соответствует адрес по которому клиент его получает, соответственно:

  есть возможность сопоставить (к пр.) JS-файлик некоторой одной страничке и только ей.

  громоздко в реализации, не оптимально с точки зрения трафика.

- **SPA (Single Page Application) -** устройство приложения, при котором за смену контента на страничке отвечает JS, при этом для определения текущего состояния используется “искуссвенный” роутинг, из этого следует, что:

  для всех станиц ассеты общие

  уменьшается объём трафика между клиентом и сервером

- **PWA (Progressive Web Application) -** устройство приложения согласно которому оно реализуется как “нативное” с возможностью оффлайн-работы и повышенной производительностью

Frontend: вспомогательные инструменты. Системы сборки. Транспиляция. Package.json. Dev серверы

JavaScript Runtime - среда исполнения JS, это может быть браузер, а может быть Node.JS - среда исполнения JS, основанная на движке V8 от Google, предоставляющая ряд возможностей, позволяющих написать backend.

Пакетные менеджеры:
- npm
- yarn

package.json - дескриптор, описывающий набор пакетов и их версий, и ряд другой информации о проекте

node_modules - папка, в которой хранятся загруженные пакеты

TypeScript - расширение JS, добавляющее статическую типизацию и всё, что с этим связано, компилируется в JS

Babel - комплиятор, занимающийся транспиляцией JS (преобразовывает код, обеспечивая совместимость новой версии JS со старыми)

Системы сборки:
- комплияция JSX
- препроцессинг стилей
- сборка ассетов
- сборка зависимостей

Webpack - популярный сборщик, интеграция с Babel, предоставляет dev-server (сервер, предоставляющий возможность увидеть изменения без полный пересборки проекта, работают на Node.JS, не используются в production)

Инструменты глобального управления состоянием и их принципы работы

Проблема шаринга одних и тех же данных с сервера между компонентами решается кэшированием при помощи глобального состояния:

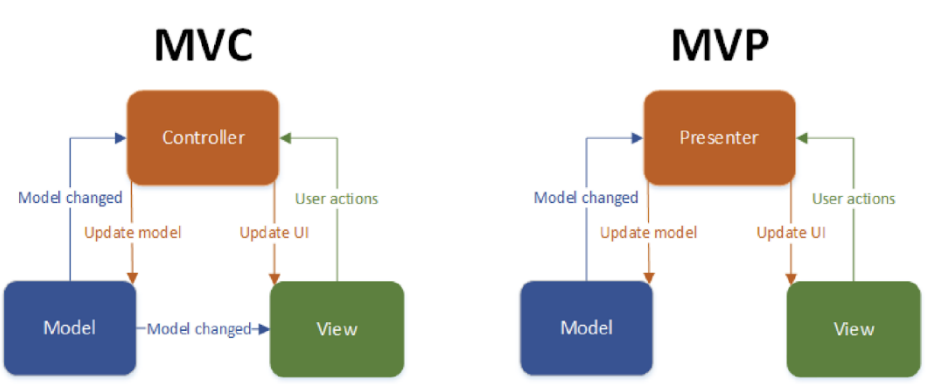
1. При получении данных разместить в глобальном состоянии по метке
2. Если данные по метке изменились или периодически, библиотека перезапросит данные с сервера

Для этого существуют библиотеки SWR, RTK Query и другие...

Развитие MVC во фронтенде. MVP. MVVM

MVC оказался недостаточно гибким, поэтому его развили в:

- **MVP** (Model-View-Presenter) - здесь Presenter реализует интерфейс для управления каким-то одним представлением через свойства и методы, позволяя создать абстракцию представления



- **MVVM** (Model-View-View Model) делает то же самое, что и MVP но без интерфейса для представления, связывание представления с view-моделью происходит автоматически

Управление состоянием (локальное vs глобальное)

Каждому компоненту присуще своё локальное состояние, однако бесполезно выполнять в каждом из них одну и ту же работу отдельно. Это решается тем, что в глобальной области видимости создаётся объект с глобальным состоянием.

REACT

React: общие принципы

React – библиотека для разработки UI интерфейсов

- Компонентный подход - смысл в переиспользовании компонент, реализующих представление и логику для некоторого элемента страницы
- Декларативность #todo (Декларативный подход)
- JSX-разметка
- Однонаправленный поток данных
- Виртуальный DOM

React: JSX

Синтаксическое расширение JavaScript, которое позволяет писать HTML-подобный код внутри JavaScript

1. Компонент возвращает 1 элемент
2. Все теги закрыты
3. Используется camelCase
4. Компилируется в JS

React: компоненты, локальное состояние пропсы

Локальное состояние — это состояние, которое управляется внутри компонента. В функциональных компонентах для работы с состоянием обычно используют хук useState.

Пропсы (или свойства) — это способ передачи данных от родительского компонента к дочернему. Они являются только для чтения и не могут быть изменены дочерним компонентом.

React: Redux Toolkit

Redux Toolkit — это официальная библиотека, упрощающая работу с Redux в React. Она предоставляет готовые функции и хуки, которые сокращают шаблонный код и делают использование Redux более интуитивным. Ключевые особенности: `configureStore`, `createSlice`, `createAsyncThunk` — для создания store, reducers и асинхронных actions соответственно. В целом, Toolkit делает Redux гораздо проще и эффективнее в использовании.

Управление состоянием (локальное vs глобальное)

* Локальное: Данные хранятся и управляются внутри отдельного компонента (например, с помощью `useState` в React). Подходит для небольших, независимых частей UI.

* Глобальное: Данные хранятся в централизованном хранилище, доступном всем компонентам приложения. Необходимо для данных, общих для многих частей приложения, или требующих синхронизации.

Инструменты глобального управления состоянием и их принципы работы

Redux: Централизованное хранилище данных, которое обновляется через `reducers`. Компоненты подписываются на изменения состояния и перерисовываются при необходимости. Обеспечивает предсказуемость и отслеживаемость изменений.

Zustand: Более легкая и простая альтернатива Redux, использующая React Hooks. Обеспечивает декларативный подход к управлению состоянием.

Recoil: Библиотека от Facebook, использующая атомы и селекторы для управления состоянием. Подходит для больших и сложных приложений, предоставляет более гибкий механизм доступа к данным, чем Redux.

Context API (React): Встроенный механизм React для обмена данными между компонентами. Может использоваться для глобального управления состоянием, но может стать громоздким в больших приложениях.

Выбор инструмента зависит от размера и сложности приложения, а также от предпочтений разработчиков. Redux Toolkit рекомендуется для больших проектов, где Redux необходим, а Zustand или Context API подходят для более мелких. Recoil может быть хорошим выбором для сложных, высокопроизводительных приложений.

ANGULAR

Фреймворк от Google

Общие понятия, архитектура

Также, как и в React используется компонентный подход, однако Angular предоставляет возможность писать HTML-шаблон, описание стилей CSS и логику как обособленно, так и в одном месте.

- Компонент объявляется декоратором @Component({}), принимающим набор свойств:
    - selector - название
    - template/templateUrl - HTML-разметка или путь к файлу с ней
    - providers - массив сервисов, данные которых использует объект
    - styles - массив путей к CSS файлам стилей
    - standalone
- Декоратор
- Директива
- Проекция контента
- Пайп
- Сервис

Декораторы

- @Component()
- @Input()
- @Output()
- @Injectable()

Директивы

Наделяют элемент поведением и в соответствии с ним, преобразуют DOM

- @if/@for/@switch + @case
- *ngIf/*ngFor
- [ngStyle]/[ngClass]

Компоненты

Жизненный цикл (их больше на самом деле):

1. OnChanges
2. OnInit
3. AfterContentInit
4. AfterViewInit
5. OnDestroy

DI

Сервисы объявленные как @Injectable() можно использовать в компонентах с помощью DI, прицины:

Инверсия управления (IoC)
Зависимости предоставляются **Angular**, а не создаются компонентами.

Singleton по умолчанию
Сервисы в **root** уровне провайдера являются одиночными (singleton).

Модульность
Сервисы можно объявлять в разных модулях или компонентах для оптимизации.

Слабое связывание
Компоненты не зависят напрямую от реализации сервиса.

Управление жизненным циклом
Angular контролирует создание и уничтожение сервисов.

ПРАКТИКА

REACT

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>})}
```

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}/>
)
```

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;
```

Регистрационная форма

```
import React, { useState } from 'react';
const Login = () => {
  const [username, setUsername] = useState('');
  const [password, setPassword] = useState('');
  const [message, setMessage] = useState('');
  const handleLogin = async (e) => {
    e.preventDefault();
    const response = await fetch('http://localhost:5000/api/login', {
      method: 'POST',
      headers: {
        'Content-Type': 'application/json',
      },
      body: JSON.stringify({ username, password }),
    });
    const data = await response.json();
    if (response.ok) {
      setMessage(data.message);
    } else {
      setMessage(data.message);
    }
  };
  return (
    <div>
      <h2>Login</h2>
      <form onSubmit={handleLogin}>
        <div>
          <label>Username:</label>
          <input
            type="text"
            value={username}
            onChange={(e) => setUsername(e.target.value)}
            required/>
        </div>
        <div>
          <label>Password:</label>
          <input
            type="password"
            value={password}
            onChange={(e) => setPassword(e.target.value)}
            required/>
        </div>
        <button type="submit">Login</button>
      </form>
      {message && <p>{message}</p>}
    </div>
  );
};

export default Login;
```

Angular

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;
  }
}
```

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
 }
}
```

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);
```

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);}}
```

CDI бины

```
@Named(value="myBean")
@ApplicationScoped / @SessionScoped / @RequestScoped
public class MyBean {
	private String text;

	public void addEnter(){
		text += "\n";
	}

	public String getText(){
		return text;
	}

	public void setText(String text){
		this.text = text;
	}
}
```

XHTML (Работа с бином)

все кидаем в <h:form>

```
<h:body>
	<h:form>
		<h1>Работа с бином про текст</h1>
		<h:otputText id="text" value="#{myBean.getText()}"/>

		<h:commandButton value="Добавить пустую строку" 
action=”#{myBean.addEnter()}”>
			<f:ajax render="text"/>
		</h:commandButton>
	</h:form>
</h:body>
```

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

```
<faces-config>
    <managed-bean>
        <managed-bean-name>myBean</managed-bean-name>
        <managed-bean-class>MyBeanClass</managed-bean-class>
        <managed-bean-scope>#{CUSTOM_SCOPE}</managed-bean-scope>
сюда можно вписать ApplicationScoped / SessionScoped / RequestScoped
    </managed-bean>
</faces-config>
```

Конфигурация web.xml, для того чтобы все запросы обрабатывал Faces Servlet по url паттерну (напрмиер всефайлы, которые закачниваются .xhtml в faces)

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

Класс для бд (JPA)

```
@Entity
@Table(name = "points")
public class PointModel {
    @Id
    private int id;

    @Column
    private double x;

    @Column
    private double y;

    ======================
```

ЭТО ПРИМЕР СО СВЯЗЬЮ (одному пользователю принадлежит много точек)

```
    @ManyToOne
    @JoinColumn(name="user_id")
    private UserModel user;
    ======================
```

```
    public PointModel(double x, double y, UserModel user) {
        this.x = x;
        this.y = y;
        this.user = user;
    }

    public PointModel(){

    }

    BCE getter/setter
}
```

```
@Entity
@Table(name = "users")
public class UserModel {
   @Id
   private int id;

   @Column
   private String email;

   @OneToMany(mappedBy="user")
   private Set<PointModel> points;

   public UserModel(int id, String email, String password) {
      this.id = id;
      this.email = email;
   }

   public UserModel(){

   }

   BCE getter/setter
}
```

Объект для подключения к БД (без конфигурации)

```
public class JpaWithoutConfigFile {
        EntityManager entityManager = null;
   public JpaWithoutConfigFile() {
```

==========

МОЖНО ПРОСТО СКАЗАТЬ ЧТО СОЗДАЕМ Map<String, String> properties СО ВСЕМИ НАСТРОЙКАМИ

```
    // Создание настроек для подключения к базе данных
    Map<String, String> properties = new HashMap<>();
    properties.put("javax.persistence.jdbc.driver", "com.mysql.cj.jdbc.Driver");
    properties.put("javax.persistence.jdbc.url",
"jdbc:mysql://localhost:3306/mydatabase");
    properties.put("javax.persistence.jdbc.user", "username");
    properties.put("javax.persistence.jdbc.password", "password");
    properties.put("hibernate.dialect", "org.hibernate.dialect.MySQLDialect");
    properties.put("hibernate.hbm2ddl.auto", "update");
    properties.put("hibernate.show_sql", "true");
```

Создание EntityManagerFactory

```
    EntityManagerFactory entityManagerFactory =
Persistence.createEntityManagerFactory("my-persistence-unit", properties);
    entityManager = entityManagerFactory.createEntityManager();
  }

  public void createNewEntity(String name){
      // Начало транзакции
    entityManager.getTransaction().begin();

    // Ваш код для работы с сущностями здесь
    // Например, создание новой сущности:
    MyEntity myEntity = new MyEntity();
    myEntity.setName(name);
    entityManager.persist(myEntity);

          // Завершение транзакции
    entityManager.getTransaction().commit();
  }
}
```