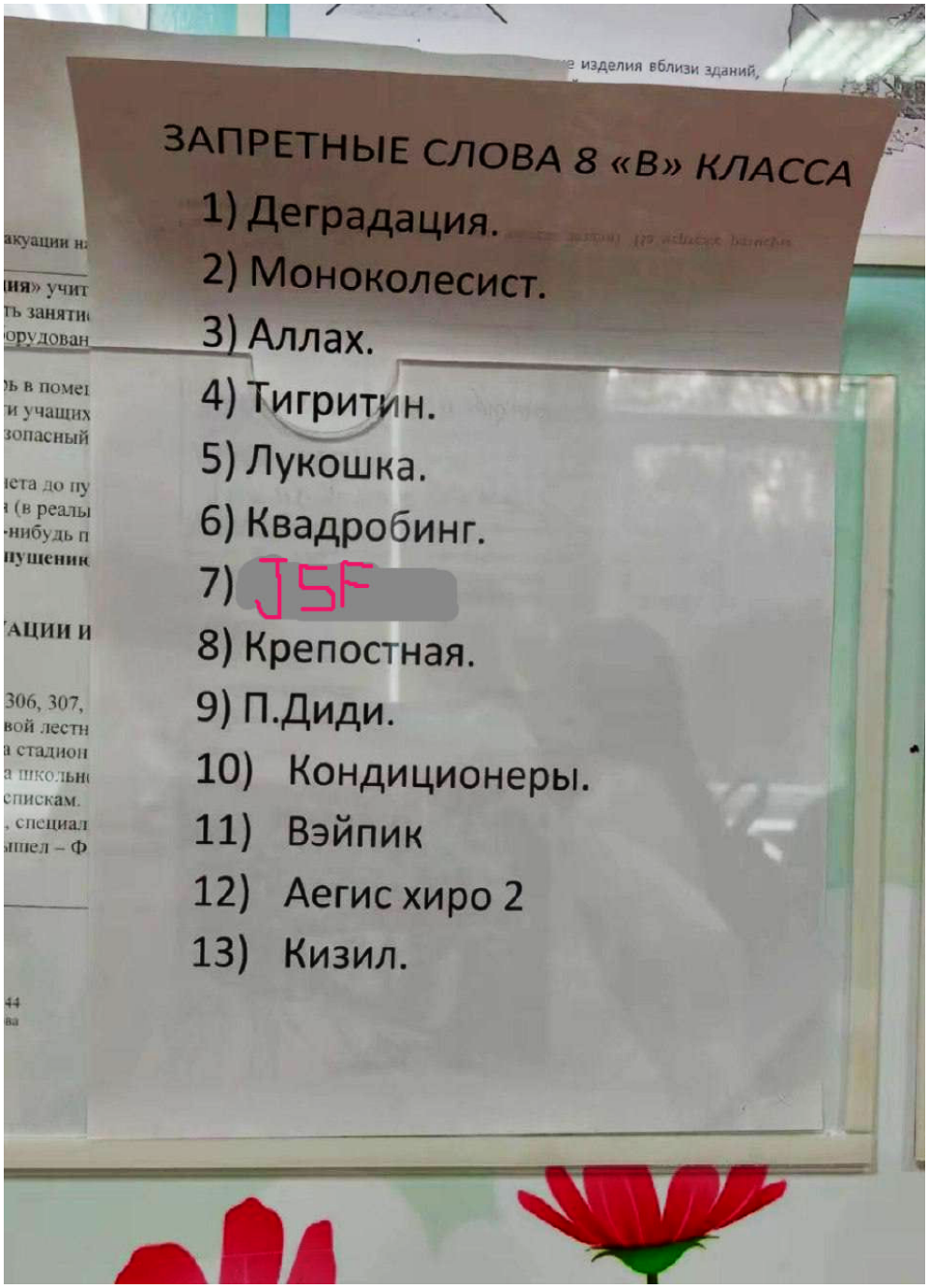
как заботать вторую рубежку (LIVE)



Подпишитесь, буду делать классный таск-трекер: github.com/FeironoX5
Колода Anki для заучивания: anki....

Оглавление

✅ 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**. Примеры контейнеров

✅ Принципы IoC, CDI, Location Transparency

Inversion **o**f **C**ontrol - принцип согласно которому компонент системы по возможности не должен полагаться на детали реализации другого её конкретного компонента, **в контексте Java EE** можно говорить об ioC-контейнере который берёт на себя **управление жизненным циклом программы** и **взаимодействием компонентов перекладывается,** снимая эти обязанности с программиста**.**
В Java EE IoC реализуется с помощью **C**ontexts and **D**ependency **I**njection - спецификации Java EE, позволяющая делать инъекции **(автоматически внедрять связи между компонентами),** управлять их **жизненным циклом** и **контекстами.**
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

✅ 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.

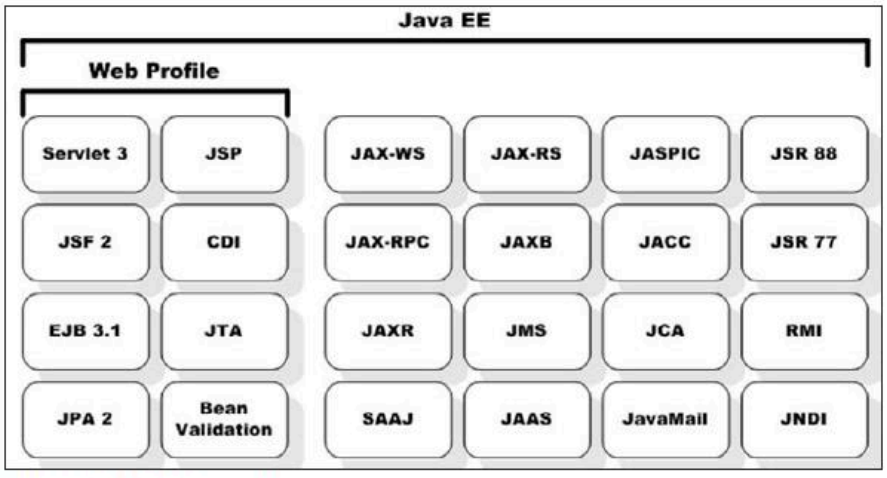


Figure 1: Java EE and the Web Profile

JSF

Источник: https://javaee.github.io/tutorial/jsf-intro.html#BNAPH

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

Bean - простой Java-класс
Контейнер JSF -
Java**S**erver **F**aces - **UI-фреймворк для создания веб-приложений**, входящий в Java EE:

- Позволяет разрабатывать веб-приложения **из UI-компонент**, связывая их с данными/моделью приложения, обрабатывать события с клиента на сервере и так далее
- Из *плюсов* **расширяемость**, **удобство** для корпоративной разработки, **широкая поддержка** среди IDE, из *минусов* **высокоуровневость** (сложно разработать собственные компоненты и работать с потоком данных напрямую)
- Реализует **паттерн MVC**:
    - Model: managed beans и прочие джава классы реализующие бизнес-логику
    - View: JSP(deprecated)/Facelets
    - Controller:

Managed beans

Бины, управляющие поведением и данные UI-компонент, используется аннотация @ManagedBean

Дерево компонентов и в принципе UI составляющая

Конфигурация

✅ Жизненный цикл

JSF Runtime управляет состоянием JSF, работа делится на 6 фаз:

1. **Restore View:**
    a. Клиент обратился в первый раз?
        i. Сформировать пустое представление
        ii. Создать объекты компонент
        iii. Назначить:
            1. Слушателей
            2. Конверторы
            3. Валидаторы
        iv. Передать в FacesContext
    b. Иначе, синхронизировать состояние компонент с клиентом
2. **Apply Request Values**
    a. Вызывать конверторы для текстовых данных пришедших с клиента
    b. Нет ошибок?
        i. Сохранить локально для компоненты в FacesContext
    c. Иначе, в FacesContext помещается сообщение об ошибке
3. **Process Validations**
    a. Вызывать валидаторы для локальных данных в компонентах из FacesContext
    b. Есть ошибки?
        i. Поместить их в FacesContext
4. **Update Model Values**
    a. Нет ошибок?
        i. Перенести данные из FacesContext в модель
5. **Invoke Application**
    a. Обработать слушателей
6. **Render Response**
    a. Создаётся/обновляется представление по результатам обработки запроса
    b. Формируется и отправляется html-страницы с ответом на запрос (ошибки из FacesContext помещаются в тег <messages>)
    c. Клиент обновляет страницу

Обработка событий

Конверторы и валидаторы

✅ CDI Beans

✅ Общее понятие

Абстрактная реализация паттерна CDI в Java EE, предполагается, что будут инъектированы

✅ Аннотации, именование

- @**Inject** в поле ниже требуется **инъектировать бин типа ...** из контейнера
- @**Named(“”) различать реализации одного интерфейса**, потом можно использовать @Inject private @Named("ca") Account account;
- **Область видимости** - определяет видимость бинов друг для друга и продолжительность их жизни:
    - @**Request**Scoped - на протяжении обработки запроса
    - @**Session**Scoped - на протяжении сессии
    - @**Application**Scoped - на протяжении жизненного цикла приложения
    - @**Conversation**Scoped - с помощью **инъекции объекта Conversation**, жизненным циклом которого **управляет программист** методами .begin() и .end()
    - @**Dependant** - по умолчанию, на протяжении жизненного цикла вызывающего

✅ "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 (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

(с презы)

🟠 Общие принципы

Спецификация, которая определяет:

- как объекты хранятся в БД
- API для работы с ними в БД
- язык запросов
- возможности использования в различных окружениях

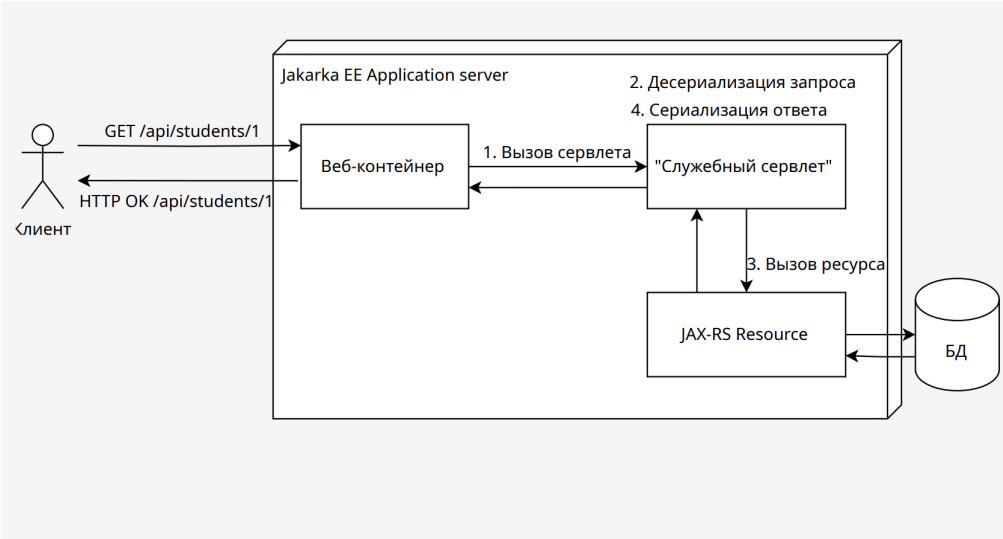
Конфигурация

🟠 Аннотации

- **@Entity** — помечает Java Bean как JPA Entity;
- **@Table** — задает название таблицы для хранения Entity;
- **@Id** — для каждой Entity обязательно должно быть поле, являющееся идентификатором;
- **@Column** — помечает поле Entity как отображаемое на столбец таблицы, можно задавать атрибуты;
- **@GeneratedValue** — на случай, если идентификатор нужно генерировать автоматически.

🟠 JAX-RS

🟠 Общие принципы



🟠 Аннотации

- **@PathParam** — достаем параметр из пути;
- **@QueryParam** — достаем URL-параметр;
- **@HeaderParam** — достаем значение заголовка;
- **@CookieParam** — достаем значение куки;
- **@FormParam** — достаем параметр из тела запроса при использовании application/x-wwwform-urlencoded запроса;
- **@Context** — достаем объекты HttpServletRequest и HttpServletResponse

✅ Frontend

✅ Рендеринг

DOM бр бр делает я хз

✅ 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

✅ 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:**

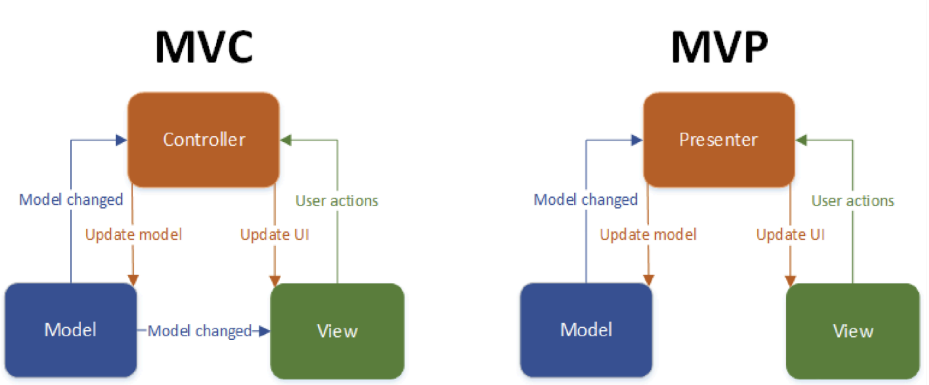
1. При получении данных **разместить в глобальном состоянии по метке**
2. Если **данные по метке изменились** или периодически, **библиотека перезапросит** данные с сервера

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

Библиотека для построения UI разрабатываемая Facebook 💩

Общие принципы

- **Компонентный подход** - смысл в **переиспользовании** компонент, реализующих представление и логику для некоторого элемента страницы
- **Декларативность** #todo
- **JSX-разметка**
- Можно писать нативные приложения

✅ JSX

Расширение JS, которое позволяет писать представление схожим с HTML образом:

- Компонент возвращает 1 элемент
- *Все* теги закрыты
- Используется camelCase
- Компилируется в JS

Компоненты, локальное состояние пропсы

Redux Toolkit

🟠 Angular

Фреймворк by 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:

- >= глобального инжектораэээээээээээ