**Лабораторная работа №3** Разработка интерфейса Web-приложения. Этап программирования.

**Цель работы:** Получить практические навыки разработки пользовательских интерфейсов на этапе применения средств разработки приложений AngularJS, ReactJS, VueJS.

**Задания к работе**

1. Установить фреймворк.
2. Разработать структуру многостраничного Web приложения – платформа электронных курсов с базой данных пользователей. Приложение должно содержать минимум: страница личного кабинета администратора, страница личного кабинета участника, страница личного кабинета создателя курсов. Включить в проект элементы взаимодействия – меню, кнопки, управляющие элементы.
3. При работе Web сайтом должна быть предусмотрен следующий функционал:

* добавление и регистрация пользователей,
* создание (удаление, редактирование) курса,
* размещение файлов курса
* расписание

**Замечание:** При условии успешной реализации данной лабораторной работы и всех предыдущих, выполненных и данных до начала зачётной недели, работа может претендовать на оценку «отлично» по дисциплине «Человеко-машинные интерфейсы»

Теоретические сведения:

### Графические редакторы: Figma, Zeplin, Adobe Photoshop

Как правило, макеты дизайна интерфейсов реализуются в таких графических редакторах, как: Figma, Zeplin или Adobe Photoshop. Поэтому владение этими инструментами, хотя бы на уровне пользователя, необходимо для грамотного переноса макетов в браузер пользователя. Разумеется, очень много зависит и от дизайнера, описывающего не только элементы интерфейса, но и их свойствами, состояния и принцип превращения, в зависимости от контекста применения (например - в условиях ограниченного пространства). Основные требования к макетам, помогающие грамотно оценить качество исполнения дизайна, будут описаны в следующих главах.

Основными манипуляциями, при работе с макетом, являются:

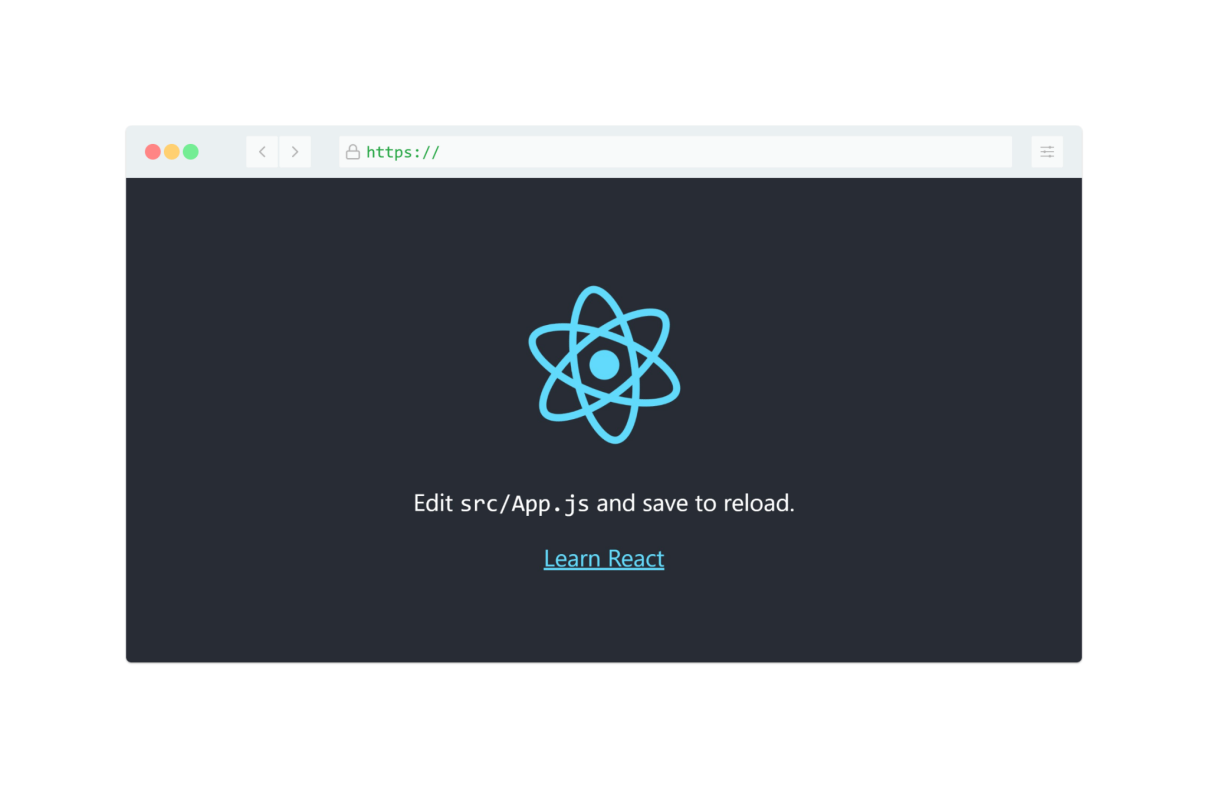
* экспорт изображений. Некоторые редакторы предоставляют возможность экспорта в разных форматах и размерах
* работа с сеткой и направляющими (линиями). Сетка помогает ориентироваться в разметке страницы, разбитой на колонки. Направляющие демонстрируют относительное выравнивание по горизонтали или вертикали. И сетка и направляющие - очень удобная, но не обязательная атрибутика макетов
* работа со слоями
* работа со свойствами элементов

## Основы фреймворка React

### 4.1. Создание и настройка готового простого SPA приложения

React - это JavaScript-библиотека для создания пользовательский интерфейсов. Для создания приложения на React нам понадобится Node.js и npm. Подробнее про установку этих программ можно узнать в пункте 1.1.

Рассмотрим способ установки приложения с помощью CLI Create React App (CRA) через npx. Убедиться, что установлена npm версии 5.2 и выше, для этого в окне терминала введите команду npm -v. Далее перейдите в директорию, где будет находится будущий проект и введите команду npx create-react-app my-project (“my-project” название приложения и директории) в окне терминала. Затем перейдем в директорию с помощью команды cd /my-project. и запустим приложение с помощью скрипта npm run start.



**Упражнения для самостоятельной работы:**

Создайте react-приложение с помощью Create React App используя npx и запустите сборку.

### 4.2. Описание структуры шаблона приложения

CRA создает приложение уже с готовой структурой. Для просмотра откройте директорию в удобном для вас IDE.  
  
 Рассмотрим развернутый с помощью CLI код:

My-project/

├── node\_modules

├── public/

│ ├── favicon.ico

│ ├── index.html

│ ├── logo192.png

│ ├── logo512.png

│ ├── manifest.json

│ └── robots.txt

├── src/

│ ├── App.css

│ ├── App.js

│ ├── App.test.js

│ ├── index.css

│ ├── index.js

│ ├── logo.svg

│ ├── reportWebVitals.js

│ └── setupTests.js

├── .gitignore

├── package-lock.json

├── package.json

└── README.md

Структура файлов react-приложения:

* *node-modules* - установленные npm-пакеты. - установленные npm-пакеты.
* *public* - содержит базовые файлы HTML, [JSON](https://www.digitalocean.com/community/tutorials/an-introduction-to-json) и изображений. Это корневые ресурсы вашего проекта.
  + *index.html -* представляет собой корневую часть вашего приложения. Именно этот файл считывает сервер, и именно его отображает ваш браузер. Откройте файл в текстовом редакторе и посмотрите на него.
  + *favicon.ico, logo192.png и logo512.png* - это значки, которые пользователь будет видеть на вкладке браузера или в телефоне. Браузер автоматически выберет значок подходящего размера. Впоследствии вы можете заменить их на другие значки, которые лучше подойдут для вашего проекта. Пока же вы можете использовать их как есть.
  + *manifest.json* - содержит структурированный набор [метаданных](https://developer.mozilla.org/en-US/docs/Mozilla/Add-ons/WebExtensions/manifest.json), описывающий ваш проект. Помимо прочего, в нем указывается, какие значки следует использовать для разных размеров экрана.
  + *robots.txt* - содержит информацию для [поисковых роботов](https://developer.mozilla.org/en-US/docs/Glossary/Robots.txt). Он указывает роботам, какие страницы можно индексировать, а какие нельзя. Вам не нужно изменять ни один из этих файлов, если у вас нет на это важных причин. Например, если вы хотите дать некоторым пользователям URL для доступа к особому контенту, который не должен быть общедоступным, вы можете добавить этот URL в файл robots.txt, и он останется общедоступным, но не будет индексироваться поисковыми системами.
* *src -* содержится код React JavaScript для вашего проекта.
  + *App.css*
  + *App.js*
  + *App.test.js*
  + *index.css*
  + *index.js*
  + *logo.svg*
  + *reportWebVitals.js*
  + *setupTests.js*
* *.gitignore* - содержит несколько директорий и файлов по умолчанию, которые система контроля исходного кода git будет игнорировать.
* *README.md* - это файл разметки, содержащий много полезной информации о приложении Create React App, в том числе обзор команд и ссылки на расширенные опции конфигурации.
* *package-lock.json* - зарезолвенные версии всех зависимостей. Нужно, чтобы у всех в команде и в CI, была одинаковая сборка. Актуально, т.к. иногда версии в package.json задаются интервалом версий.
* *package.json* - список зависимостей проекта, а также список настроенных скриптов.

### 4.3. Компоненты

Компоненты позволяют разбить интерфейс на независимые части. Их можно складывать вместе и использовать несколько раз. Во многом компоненты ведут себя как обычные функции JavaScript. Они принимают произвольные входные данные (так называемые «пропсы») и возвращают React-элементы, описывающие, что мы хотим увидеть на экране. В React компоненты могут быть функциональными или классовыми.   
Функциональные компоненты строятся на базе функци. Принимают параметры (“пропсы”) и возвращают React-элемент.

import React from "react"

function MyComponent() {

return <span>It's my component</span>

}

export default MyComponent;

Классовые компоненты создаются на основе классов.

import React, { Component } from "react"

class MyComponent extends Component {

render() {

return <span>It's my class component</span>

}

}

export default MyComponent

Функциональным и классовым компонентам доступны дополнительные возможности. Подробнее о компонентах можно узнать в документации React <https://ru.reactjs.org/docs/components-and-props.html#gatsby-focus-wrapper>.  
 В данной методичке будет рассмотрены только функциональные компоненты и их возможности.  
  
 Рассмотрим части компонента.  
  
 **import React from "react"**

Импортируется объект React для работы с jsx. Подробнее про jsx можно узнать здесь <https://ru.reactjs.org/docs/jsx-in-depth.html>  
 **function MyComponent() {**  
 Объявляется функция, идентификатор которого будет название компонента.  
 **return <span>It's my component</span>**  
 Функция возвращает React-элемент

**export default MyComponent**   
Экспортирование функций для подключения в другие файлы.  
**import MyComponent from "./components/my-component"**

**import "./App.css"**

**function App() {**

**return (**

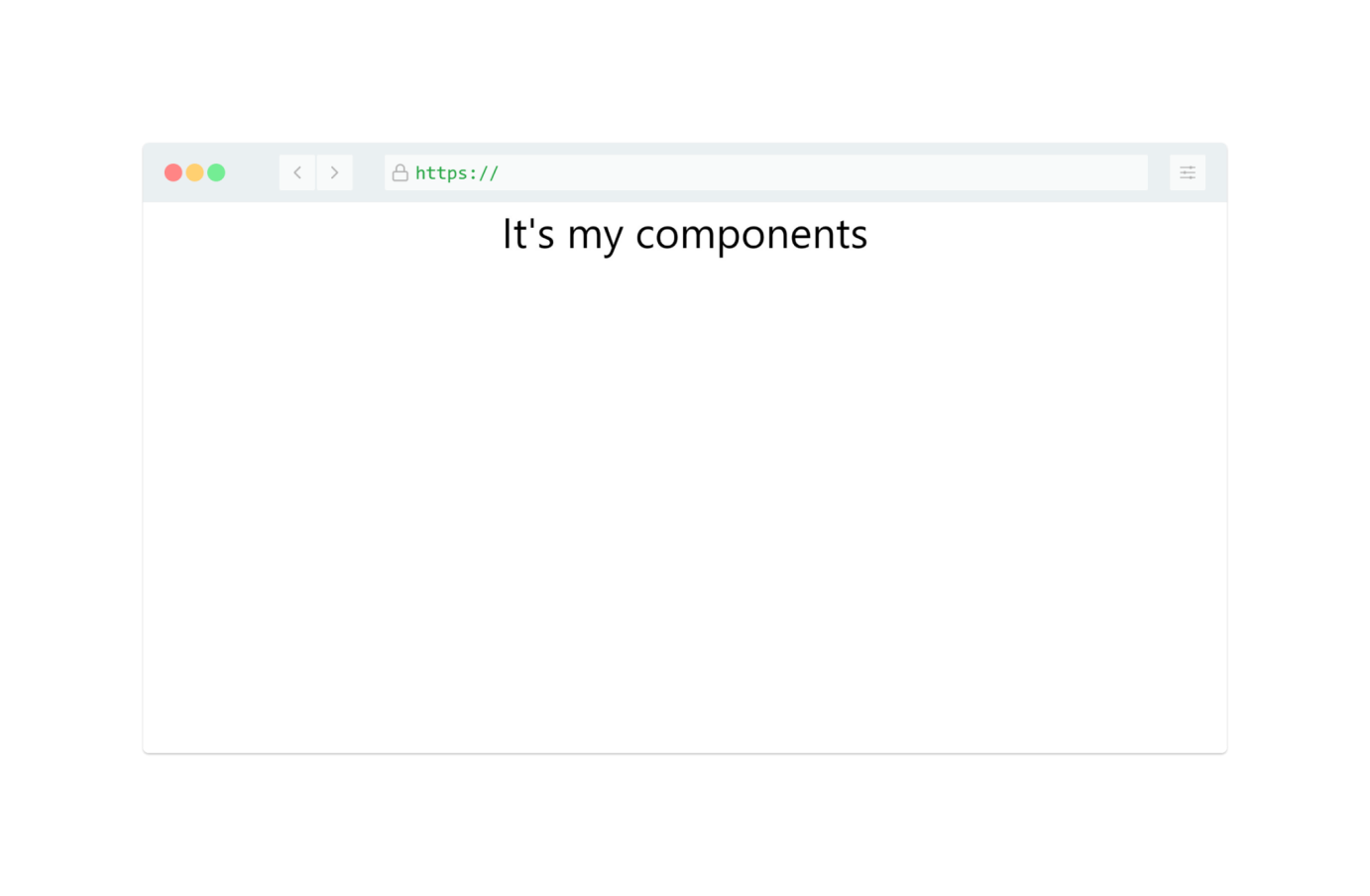
**<div className="App">**

**<MyComponent />**

**</div>**

**)**

**}**

**export default App**Импортирование компонента в файл App.js  


В компоненты можно подключать другие компоненты.

**import React from "react"**

**function ChildComponent() {**

**return <div>It's child component</div>**

**}**

**export default ChildComponent**

Компонент ChildComponent, который будет подключен в MyComponent

**import React from "react"**

**import ChildComponent from "./child"**

**function MyComponent() {**

**return (**

**<div>**

**<span>It's my components</span>**

**<ChildComponent />**

**</div>**

**)**

**}**

**export default MyComponent**

**import ChildComponent from "./child"**

Импортируется компонент ChildComponent

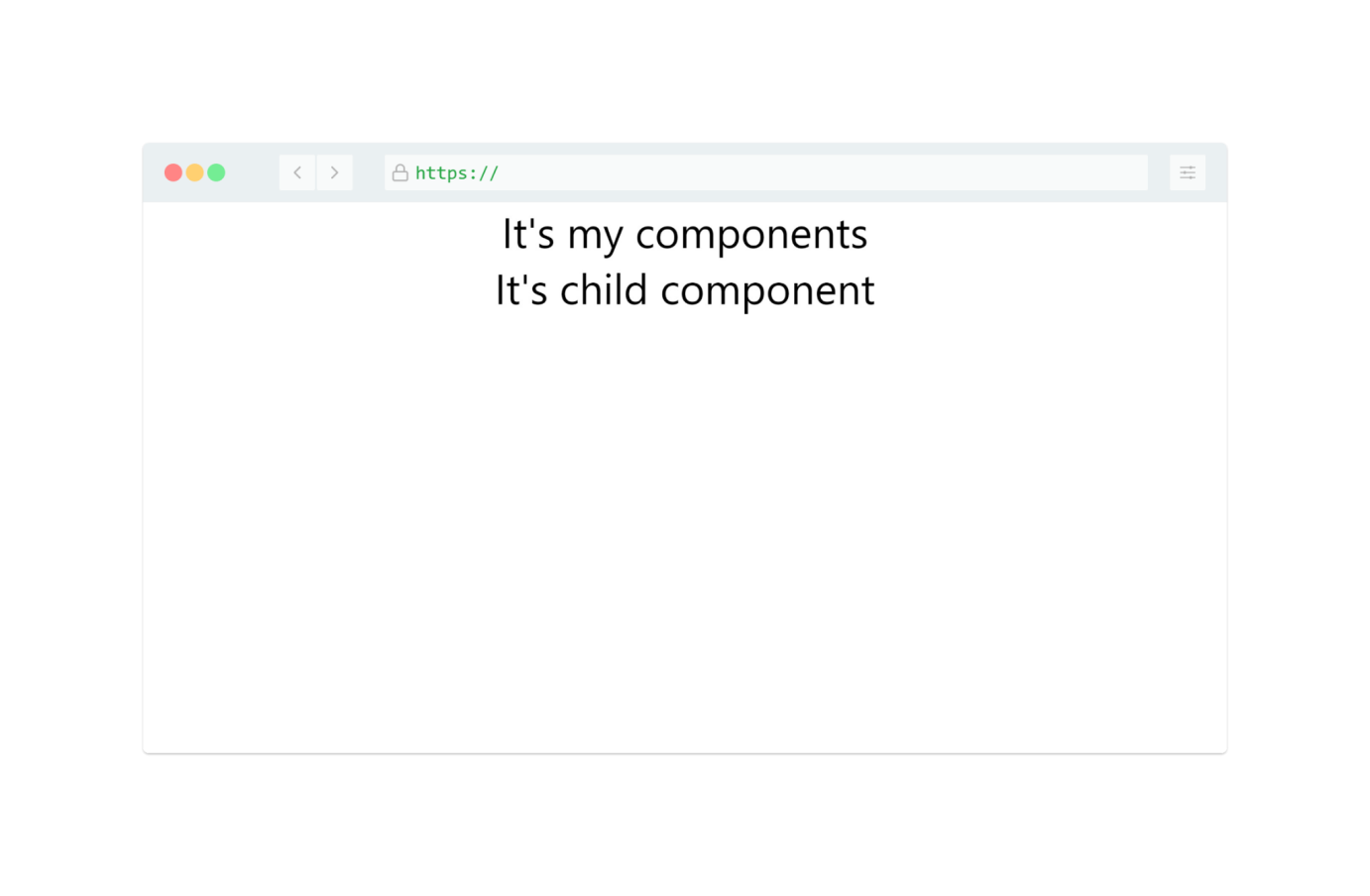
**<div>**

**<span>It's my components</span>**

**<ChildComponent />**

**</div>**

Добавляем компонент в разметку



У компонентов всегда должна быть родительская обертка.

**<div>**

**<span>It's my components</span>**

**<ChildComponent />**

**</div>**

Элемент div является родительской оберткой.

**function MyComponent() {**

**return (**

**<span>It's my components</span>**

**<ChildComponent />**

**)**

**}**

Если убрать тег div, будет вызвана ошибка: “JSX expressions must have one parent element”.

Чтобы не создавать лишних узлов нужно использовать фрагменты. Подробнее про фрагмент можно узнать в этой статье <https://reactdev.ru/handbook/fragments/>.

**function MyComponent() {**

**return (**

**<>**

**<span>It's my components</span>**

**<ChildComponent />**

**</>**

**)**

**}**

В React удобно работать с повторяющимися элементами.

**import React from "react"**

**const users = [**

**{**

**id: 1,**

**name: "Alex",**

**},**

**{**

**id: 2,**

**name: "Adam ",**

**},**

**{**

**id: 3,**

**name: "Jason ",**

**},**

**{**

**id: 4,**

**name: "Eric ",**

**},**

**{**

**id: 5,**

**name: "Scott ",**

**},**

**]**

**function List() {**

**return (**

**<ol>**

**{users.map((user) => (**

**<li key={user.id}>{user.name}</li>**

**))}**

**</ol>**

**)**

**}**

**export default List**

Компонент List выводит имена пользователей

**const users = [**

**{**

**id: 1,**

**name: "Alex",**

**},**

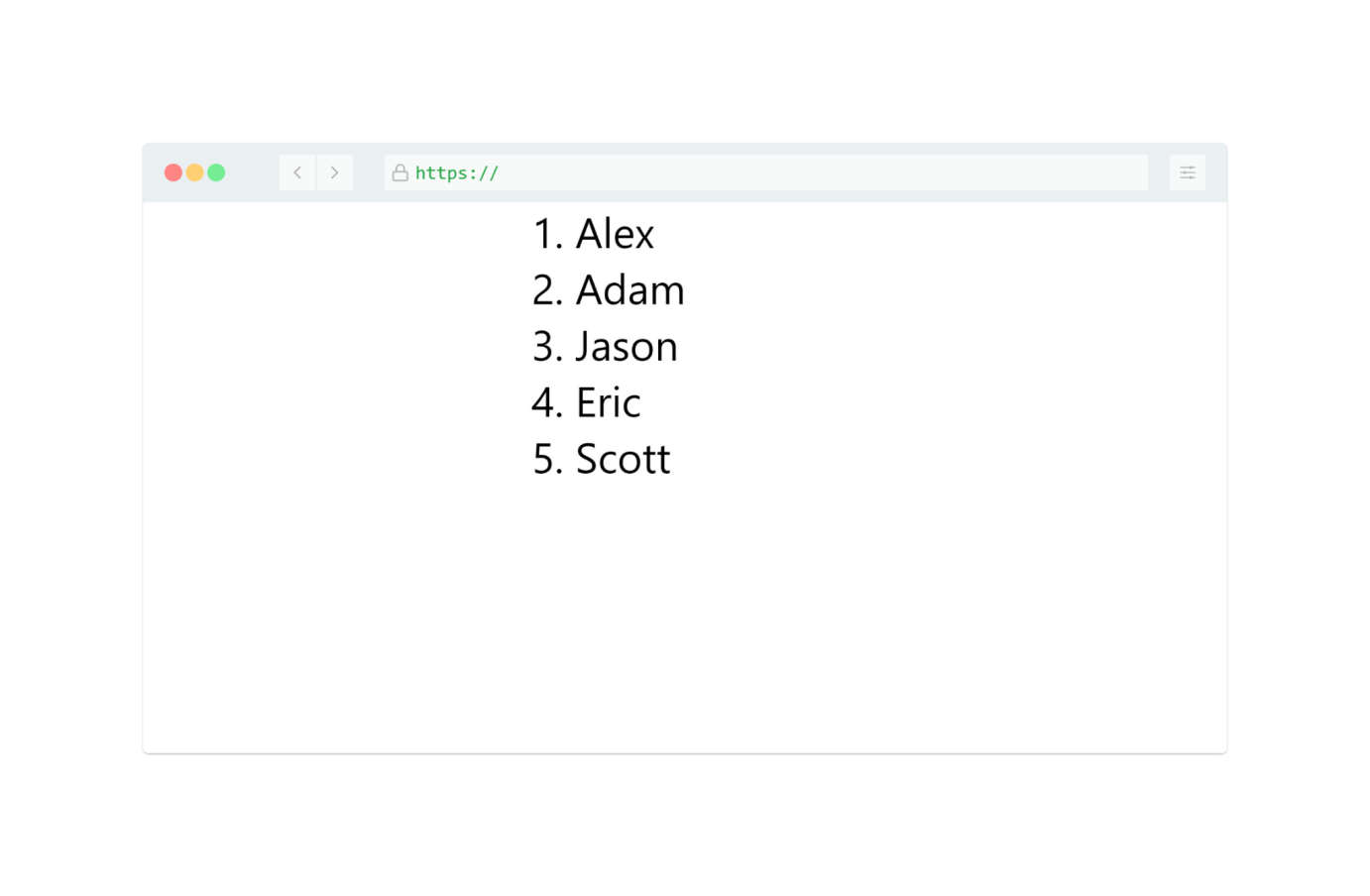
Элементами массива users являются объекты, у которых два поля id - уникальный идентификатор пользователя и name - имя пользователя.

**{users.map((user) => (**

**<li key={user.id}>{user.name}</li>**

**))}**

У массива users вызывается метод map, который по каждой итерации возвращается react-элемент. При таком подходе у возвращаемого тега должен быть обязательный атрибут key, которому присваивается уникальное значение (чаще всего id). Подробнее про key можно узнать в статье <https://ru.reactjs.org/docs/reconciliation.html#keys>



**Упражнения для самостоятельной работы**  
Создайте компонент, который будет выводить ваше имя в заголовке.   
Импортируйте новый компонент в App.js файл.

### 4.4. Пропсы

Пропсы (props) - это аргументы, передаваемые в React компонент. Пропсы служат только для чтения, их нельзя изменять.

**import React from "react"**

**function User(props) {**

**return <h1>Hello, {props.name}</h1>**

**}**

**export default User**

Компонент User, который рендерит имя пользователя.

**function User(props) {**

Аргумент props - это объект, который содержит все передаваемые значения в компонент.  
**return <h1>Hello, {props.name}</h1>**

Выводим имя пользователя, которое передали в пропс name.

**import User from './components/user';**

**import "./App.css"**

**function App() {**

**return (**

**<div className="App">**

**<User name="Alex" />**

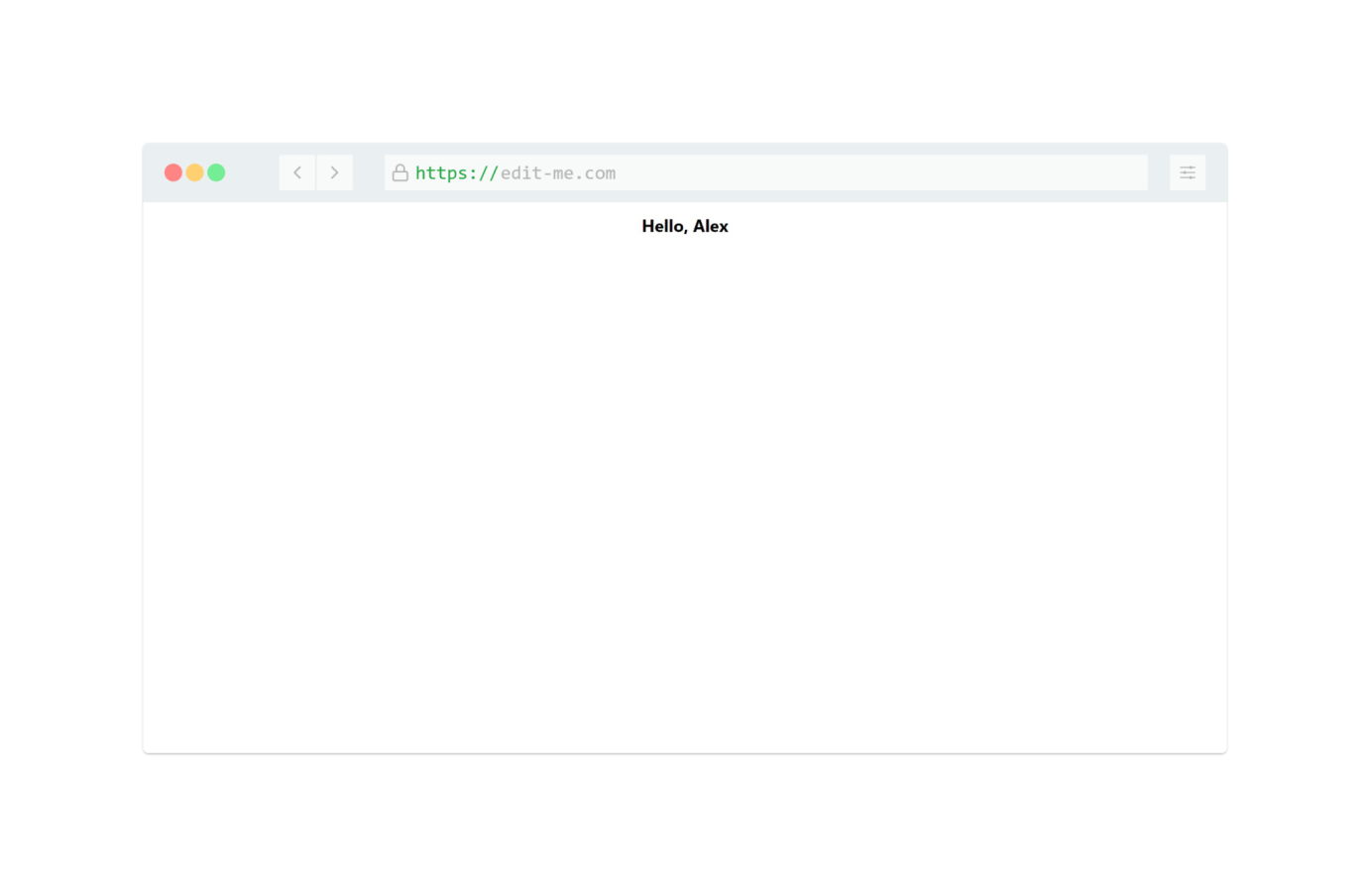
**</div>**

**)**

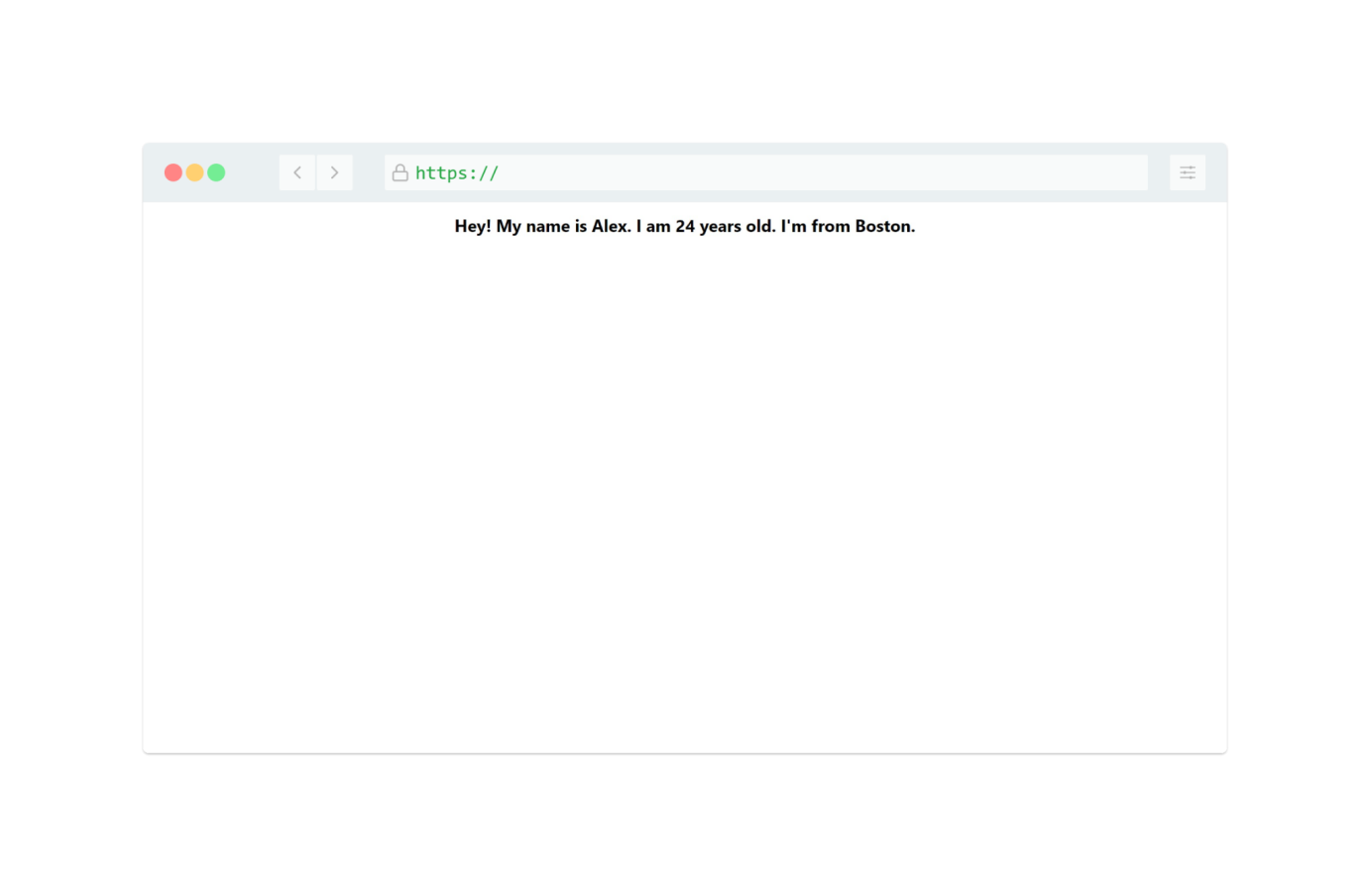
**}**

**export default App**

Импортировали компонент User и передали имя в аргумент name.



**Упражнения для самостоятельной работы**  
 Создайте новый компонент для отображения имя пользователя, возраст и город.



### 4.5. Жизненный цикл и хуки

У компонента React есть жизненный цикл, который позволяет выполнять различные действия: получение данных с сервера, подписка и отписка событий и другое.

Подробнее про циклы можно узнать <https://dev-gang.ru/article/kak-ponjat-metody-zhiznennogo-cikla-komponenta-v-reactjs-m3v6725v7q/>  
 Если в классовых компонентах были жизненные методы, то в функциональных компонентах используются хуки (<https://ru.reactjs.org/docs/hooks-intro.html>)  
 Для изменения состояния компонента используется хук useState  
 **import React, { useState } from "react"**

**function Count() {**

**const [count, setCount] = useState(0)**

**return (**

**<div>**

**<h1>Count of click {count}</h1>**

**<button**

**onClick={() => {**

**setCount(count + 1)**

**}}**

**>**

**Click Me**

**</button>**

**</div>**

**)**

**}**

**export default Count**

Компонент отображает количество нажатий на кнопку.  
**import React, { useState } from "react"**

Импортируется хук useState

**const [count, setCount] = useState(0)**

Деструктуризируем возвращаемый массив из функции useState. Первый элемент массива - это состояние под идентификатором count, второй элемент - это функция для изменения состояния setCount. Сам хук принимает первоначальное состояние, в нашем случаи это 0;  
  
 **<h1>Count of click {count}</h1>**

Выводим в заголовке количество кликов.

**<button**

**onClick={() => {**

**setCount(count + 1)**

**}}**

**>**

**Click Me**

**</button>**

На событие onClick для элемента button изменяется состояние count.  
 Подробнее про события в React можно узнать в статье <https://ru.reactjs.org/docs/handling-events.html>

Изменение состояния через дочерний компонент.

import React, { useState } from "react"

import ChildComponent from "./child"

**function MyComponent() {**

**const [count, setCount] = useState(0)**

**return (**

**<>**

**<h1>Click of count: {count}</h1>**

**<ChildComponent onChangeCount={setCount} />**

**</>**

**)**

**}**

**export default MyComponent**

Компонент отображает количество кликов.

**<>**

**<h1>Click of count: {count}</h1>**

**<ChildComponent onChangeCount={setCount} />**

**</>**

У компонента ChildComponent есть пропс onChangeCount, в которой передается функция для изменения состояния

**import React from "react"**

**function ChildComponent(props) {**

**return (**

**<button onClick={() => props.onChangeCount((prevCount) => prevCount + 1)}>**

**Click me**

**</button>**

**)**

**}**

**export default ChildComponent**

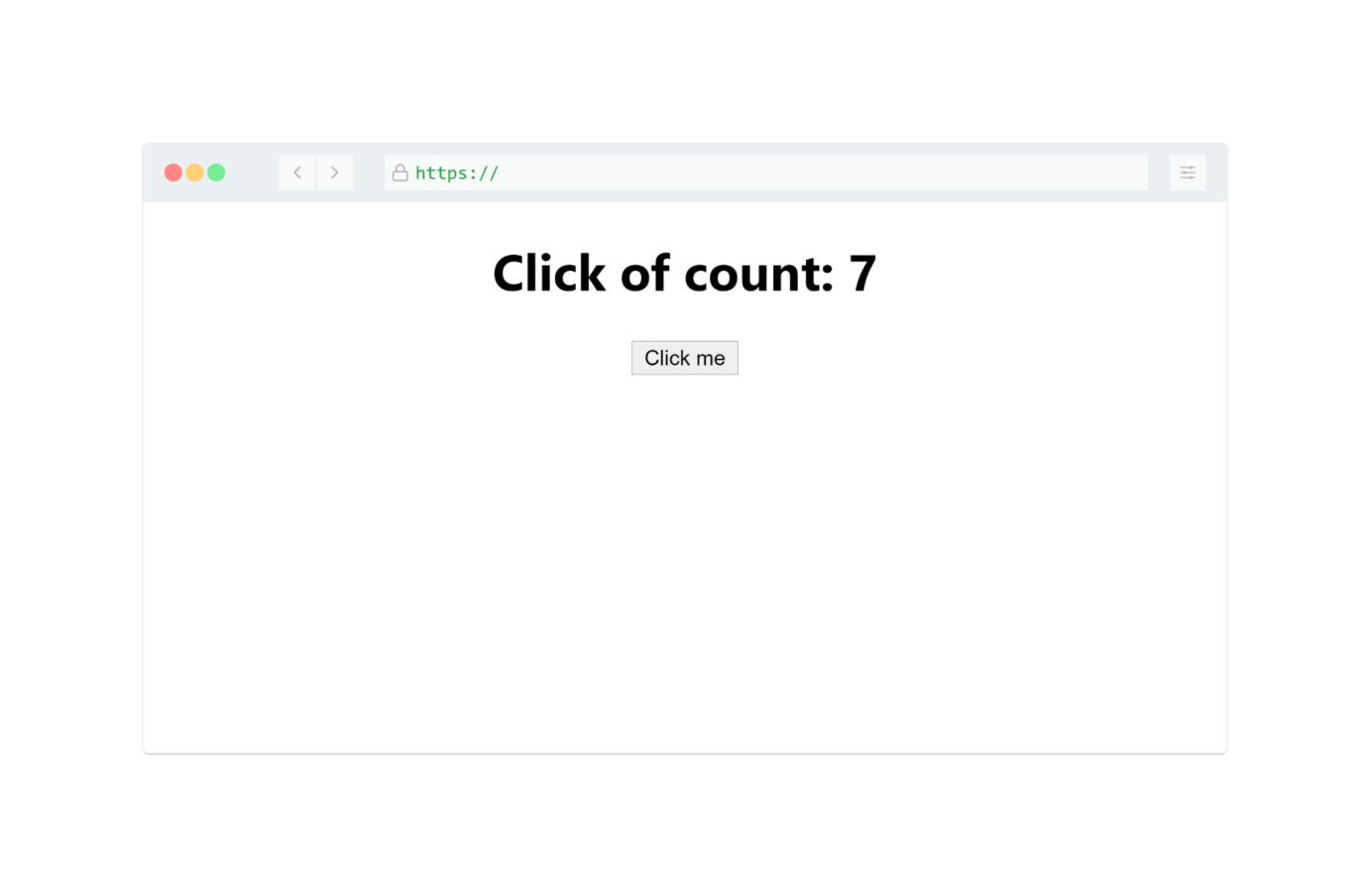
ChildComponent изменяет состояние count при нажатие на кнопку.

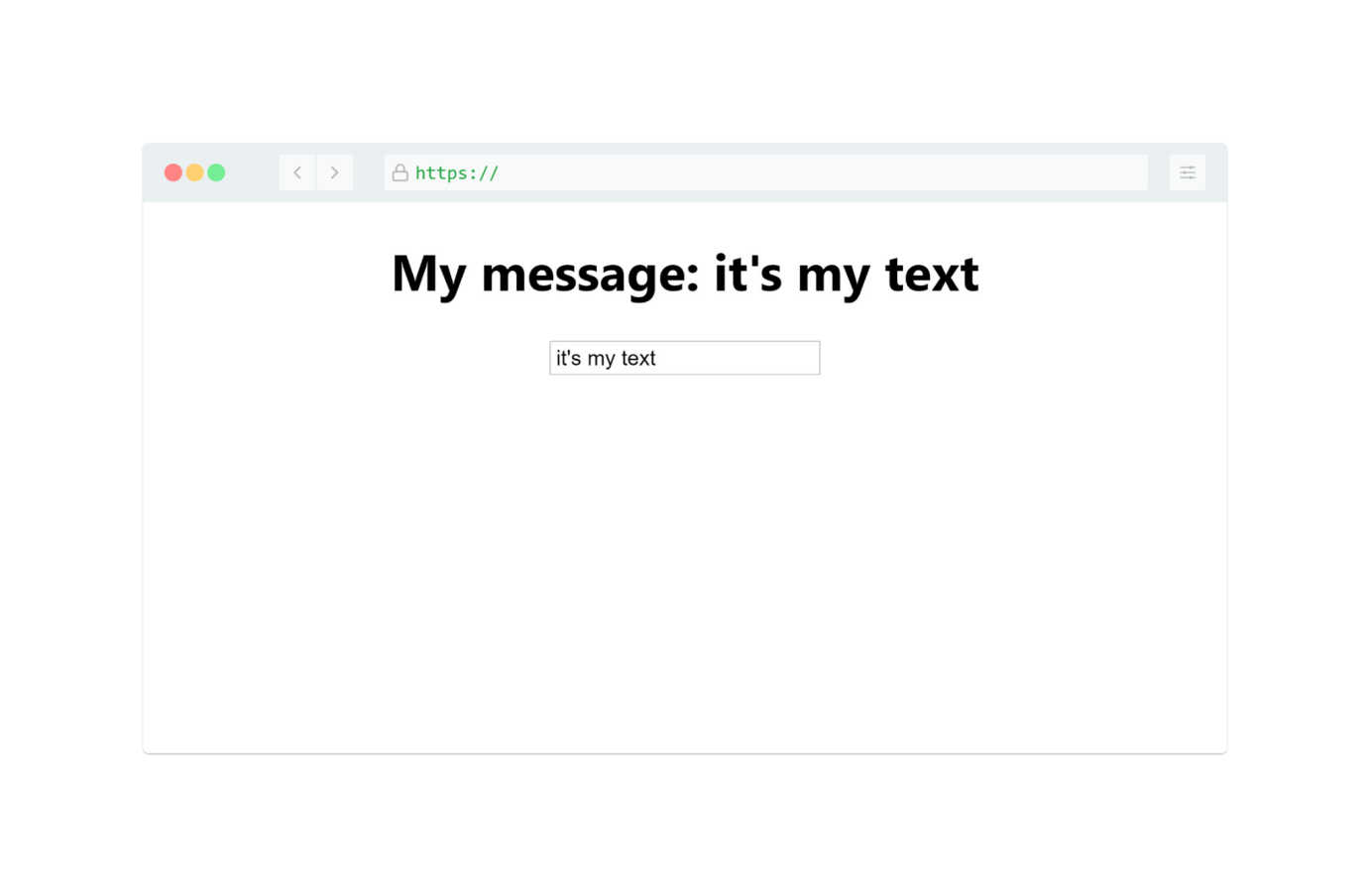
**button onClick={() => props.onChangeCount((prevCount) => prevCount + 1)}>**

**Click me**

**</button>**

При нажатие на кнопку срабатывает событие onClick, которая вызывает функция onChangeCount, в которую передали setCount. Если аргументом setCount будет функция, то аргументом этого коллбэка будет предыдущее состояние - prevCount, а возвращаться будет новое значение состояния count.





Хук useEffect

Данный хук предназначен для имитирования жизненных методов в функциональном компоненте. Подробнее об useEffect можно узнать <https://ru.reactjs.org/docs/hooks-reference.html#useeffect>

**import React, { useEffect } from "react"**

**function Log(props) {**

**useEffect(() => {**

**console.log("Component mounted")**

**return () => {**

**console.log("Component unmounted")**

**}**

**}, [])**

**useEffect(() => {**

**console.log("Props name changed")**

**}, [props.name])**

**return <h1>My name is {props.name}</h1>**

**}**

**export default Log**

Компонент Log выводит в консоли браузера сообщения, что компонент смонтирован, размонтирован и выводит сообщение, если пропс name изменился.

**import React, { useEffect } from "react"**

Импортируется хук useEffect  
**useEffect(() => {**

**console.log("Component mounted")**

**return () => {**

**console.log("Component unmounted")**

**}**

**}, [])**

Объявляется функция useEffect, которая первым аргументом принимает коллбэк, вторым - массив зависимостей. Массив зависимостей пустой , поэтому хук сработает только при монтирование компонента. Если коллбэк возвращает функцию, то данная функция будет вызвана при размонтирование коллбэка.

**useEffect(() => {**

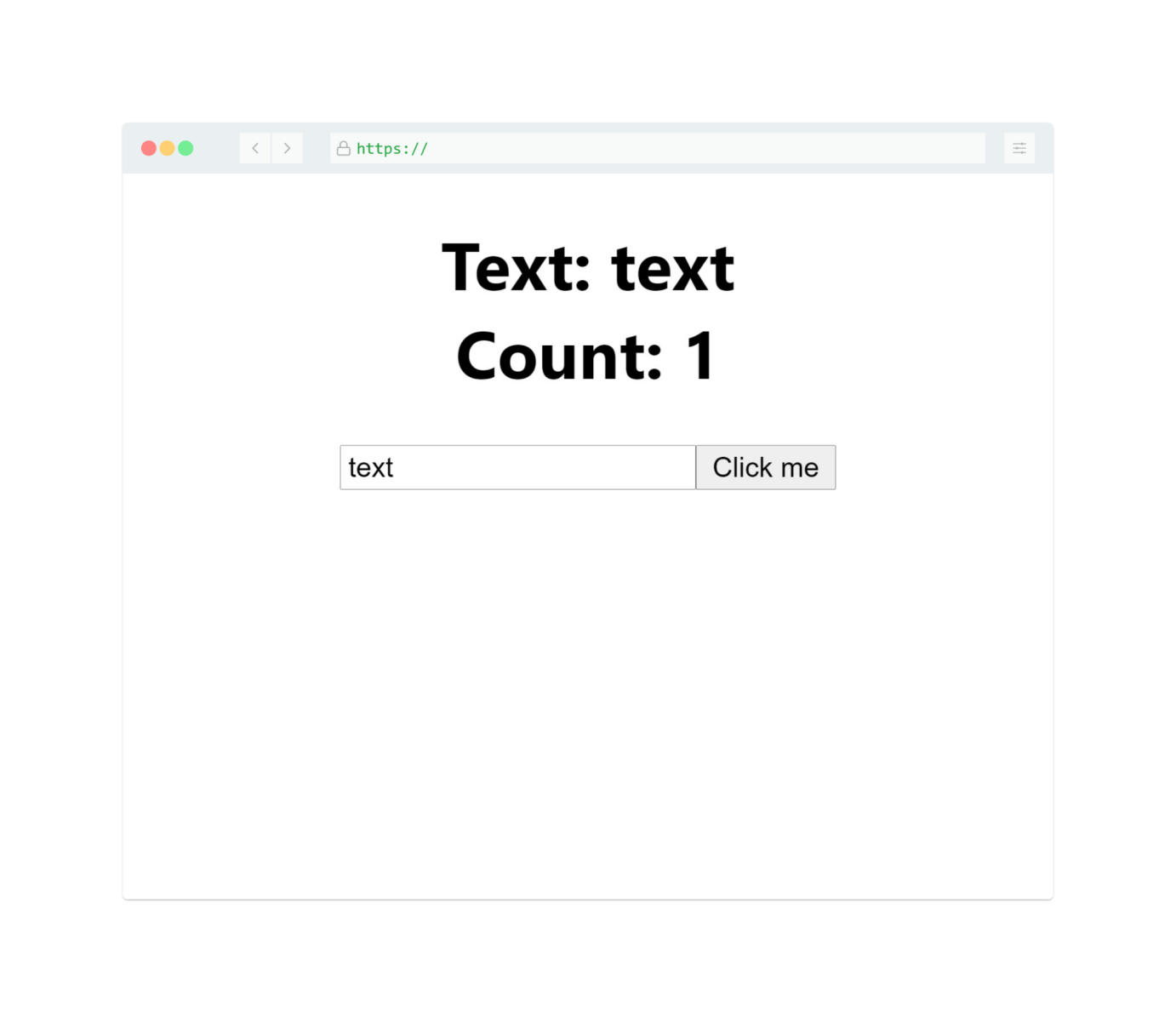
**console.log("Props name changed")**

**}, [props.name])**

Данный хук будет выводить сообщение в консоль при изменение пропса name, который был передан в массив зависимостей.

**Упражнения для самостоятельной работы**

Создайте компонент, который в консоли браузера будет выводить общее количество рендеров. Добавьте два состояния text и count. Состояние count изменяется при нажатие на кнопку, а text при вводе символов в поле ввода.



### 4.6. Контекст

Контекст позволяет передавать данные через дерево компонентов без необходимости передавать пропсы на промежуточных уровнях. Подробнее об контексте можно узнать <https://ru.reactjs.org/docs/context.html>

**import { createContext } from "react"**

**const Context = createContext("")**

**export default Context**

Файл context.js\*-

**import { createContext } from "react**"

Импортируется createContext для создания контекста

**const Context = createContext("")**

Объявляется переменная с идентификатором Context и присваивается результат функции createContext, которая принимает дефолтное значение контекста. В данном случае пустая строка.

**export default Context**

Экспортируется переменная Context.

**import React, { useState } from "react"**

**import Child from "./child"**

**import Context from "./context"**

**function Parent() {**

**const [text, setText] = useState("")**

**return (**

**<div>**

**<input value={text} onChange={(e) => setText(e.target.value)} />**

**<Context.Provider value={text}>**

**<Child />**

**</Context.Provider>**

**</div>**

**)**

**}**

**export default Parent**

Компонент Parent передает состояние text в контекст.

**import Context from "./context"**

Импортируется контекст

**<Context.Provider value={text}>**

**<Child />**

**</Context.Provider>**

Оборачивает компонент Child контекстом. У контекста вызывается свойство Provider, c аргументом value. В value присваивается значение состояние text.

import React from "react"

import SecondChild from "./second-child"

**function Child() {**

**return (**

**<div>**

**It's First Child**

**<SecondChild />**

**</div>**

**)**

**}**

**export default Child**

Компонент Child отображает компонент SecondChild.

**import React, { useContext } from "react"**

**import Context from "./context"**

**function SecondChild() {**

**const text = useContext(Context)**

**return <div>Text of second child: {text}</div>**

**}**

**export default SecondChild**

Компонент SecondChild отображает значение состояния text.

**import React, { useContext } from "react"**

Импортируется хук useContext для получение значения, которое передали в свойство value.

**const text = useContext(Context)**

Присвоение значение контекста в переменную text.

**return <div>Text of second child: {text}</div>**

Вывод значения состояния text.

## Глава 4. Основы фреймворка Angular

### 4.1. Создание и настройка готового простого SPA приложения

Для работы Angular требует последнюю LTS версию node.js[[1]](#footnote-1). Если пункт 1.1. выполнен (для удобства рекомендуется использовать nvm) и настроен git (email и name пользователя), то можно перейти к установке angular CLI[[2]](#footnote-2). CLI достаточно мощный и с помощью него можно заскафолдить многое, как мы увидим далее.

Устанавливаем глобально CLI:

$ npm install -g @angular/cli

Теперь нам доступна утилита ng.

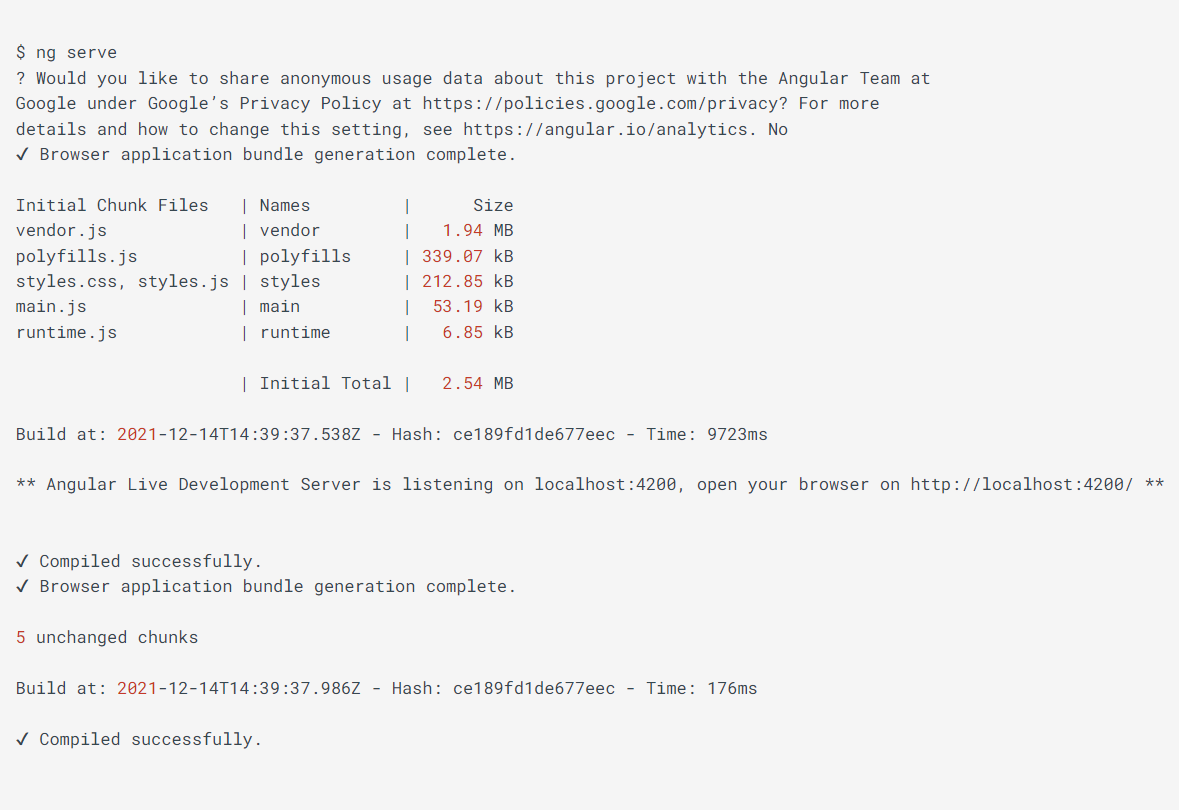
Вводим команду $ ng new my-app[[3]](#footnote-3)



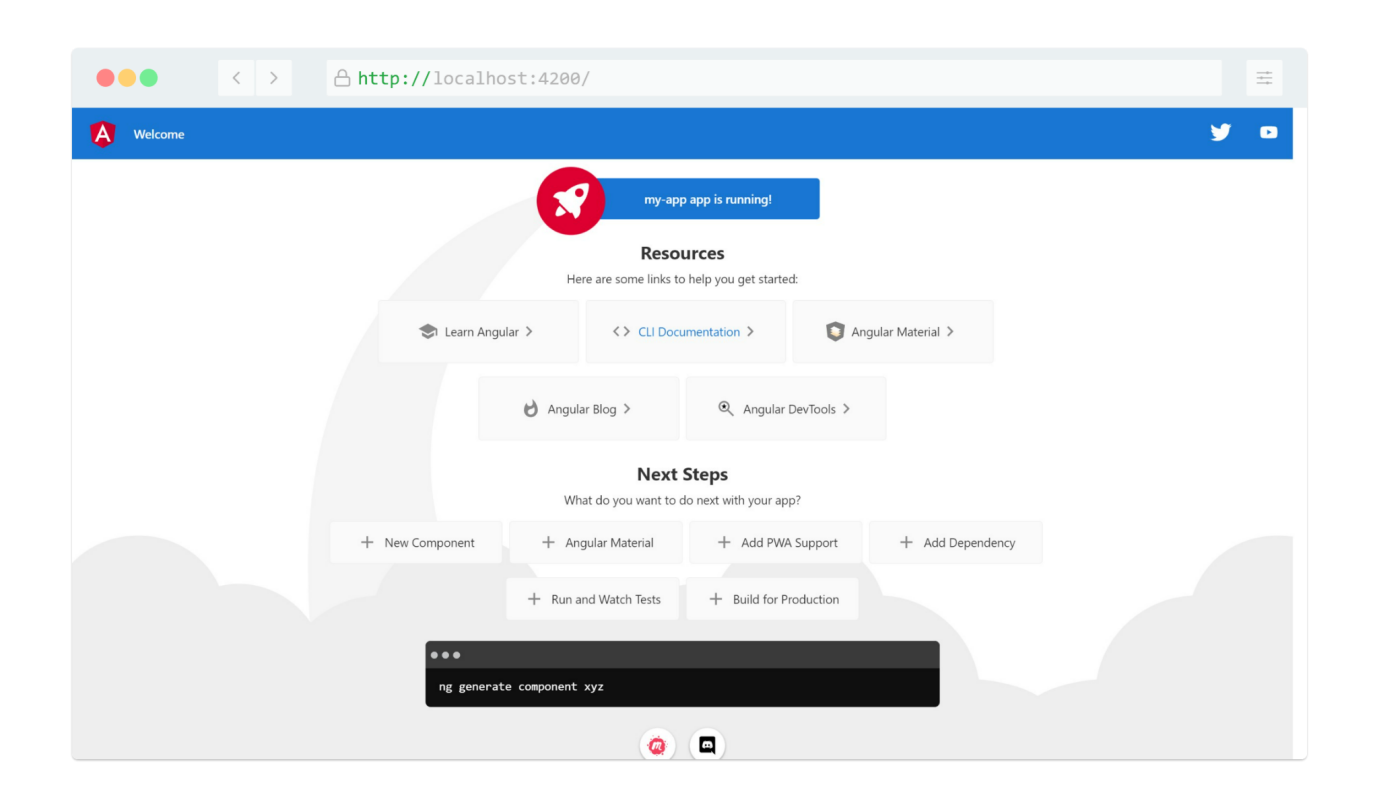
В интерактивном режиме будут заданы вопросы, нужно ли сразу заложить модуль для роутинга, какой препроцессор стилей использовать.

Развертывание приложения может также настраивать предварительно передав нужный флаг. Например, можно указать иной пакетный менеджер (не npm), или развернуть пустой репозиторий, чтоб потом туда заскфаволдить несколько приложений, получив на выходе монорепозиорий.

На этом развертывание закончено. Теперь можно запускать командой $ ng serve[[4]](#footnote-4) из директории с проектом.



По дефолту проект поднимается на порту 4200, поэтому в браузере открываем localhost:4200. Где можно наблюдать сгенеренный корневой компонент с контентом.



Несмотря на полную готовность к работе, это сложно назвать SPA приложением, ведь у нас всего одна страница. Мы могли в начале заскафолдить роутер модуль, но несколько страниц не будет развернуто средствами CLI. Поэтому см пункт 1.13.8, где будет разобран роутинг в ангуляре.

**Упражнения для самостоятельной работы:**

Разверните приложение на ангуляре и запустите его по инструкции выше.

### 4.2. Описание структуры шаблона приложения

Рассмотрим развернутый с помощью CLI код:

my-app/

├── node\_modules/

│ └── ...

├── src/

│ ├── app/

│ │ └── ...

│ ├── assets/

│ │ └── ...

│ ├── environments/

│ │ └── ...

│ ├── favicon.ico

│ ├── index.html

│ ├── main.ts

│ ├── polyfills.ts

│ ├── styles.scss

│ └── test.ts

├── .browserslistrc

├── .editorconfig

├── .gitignore

├── angular.json

├── karma.conf.js

├── package-lock.json

├── package.json

├── README.MD

├── tsconfig.json

├── tsconfig.app.json

└── tsconfig.spec.json

Разберем структуру файлов на верхнем уровне

* *node\_modules* - установленные npm-пакеты.
* *src* - код проекта. В качестве альтернативы при разворачивании можно вместо src развернуть папку projects, в которой можно потом заскафолдить произвольное количество проектов.
  + *app* - основная кодовая база проекта (рассмотрим далее)
  + *assets* - ассеты проекта, картинки, шрифты и иные вспомогательные файлы которые нужны как есть. При сборке данная папка будет скопирована в папку с готовым билдом.
  + *environments* - различные переменные в виде ts-файла. На уровне dev и prod сборок подкладываются разные файлы. Можно точечно конфигурировать работу приложения под разные демо-стенды.
  + *favicon.ico* - иконка приложения на панели вкладок в браузере, по умолчанию с изображением ангуляра, в реальном проекте заменяется на иконку проекта.
  + *index.html* - корневой файл с версткой, является основой для итогового файла который будет отдаваться бекендом на любой урл фронта. Сюда можно вставить дефолтную верстку (например, прелодер, который скрасит время ожидания пользователя до запуска приложения пока грузиться бандл).
  + *main.ts* - точка входа в приложения, главный файл, который запускает приложение на Angular.
  + polyfills.ts - файл для подключения полифилов, чтобы была поддержка более старых версий браузеров.
  + *styles.scss* - файл с глобальными стилями css.
  + *test.ts* - вспомогательный файл для тестов.
* *.browserslistrc* - список поддерживаемых браузеров. Используется вебпаком при сборке скриптов и стилей, для генерации минимального кода с максимальной поддержкой требуемых браузеров.
* *.editorconfig* - файл с ключевыми требованиями по стилизации кода. Данный файл поддерживается многими редакторами, переопределяя глобальные настройки. Нужно для единообразия кода во всей команде.
* *.gitignore* - список файлов, которые не нужно отслеживать гитом (например, node\_modules, папки с конфигами ide, папка с билдом при локальной сборке).
* *angular.json* - основной конфиг сборок. Можно настроить разные сборки для разных флагом сборок, подключать файлы в сборку, перекладывать файлы при сборке, настраивать локализацию сборок и тд.
* *karma.conf.js* - основной конфиг тестраннера.
* *package-lock.json* - зарезолвенные версии всех зависимостей. Нужно, чтобы у всех в команде и в CI, была одинаковая сборка. Актуально, т.к. иногда версии в package.json задаются интервалом версий.
* *package.json* - список зависимостей проекта, а также список настроенных скриптов.
* *README.MD* - корневой файл с документацией разработчика. Сюда можно написать информацию по всему проекту, как разворачивать, собирать, и тд.
* *tsconfig.json* - корневой файл с конфигурацией тайпскрипта всего репозитория.
* *tsconfig.app.json* - конфигурация для проекта. Наследуется от корневого файла. Если в репозитории развернуто несколько проектов, то будет по файлу на каждый проект.
* *tsconfig.spec.json* - конфигурация для тестов, наследуется от корневого.

Рассмотрим теперь содержимое папки *src/app*:

my-app/src/app/

├── app.module.ts

├── app.component.ts

├── app.component.scss

├── app.component.html

├── app.component.spec.ts

└── app-routing.module.ts

* app.module.ts - ангуляр модуль. Angular имеет свою модульную архитектуру, которая зародилась еще в angular.js, во времена es5, когда один файл не являлся сам по себе модулем как в es6. В современном Angular применяется также с целью создать области видимости компонентов, а также включить в бандл их, провайдеры и пайпы. С появлением *provideIn* в декораторах провайдеров и *standalone* компонентов роль модулей начала отходить на задний план.
* *app-routing.module.ts -* модуль с настройками *RouterModule*, чтобы работала навигация в приложении. Генерируется, если был выбран нужный вариант при генерации проекта.
* *app.component.\** - файлы корневого компонента (стили, логика, верстка и тесты). Компоненты рассмотрим подробнее в пункте 1.13.3.

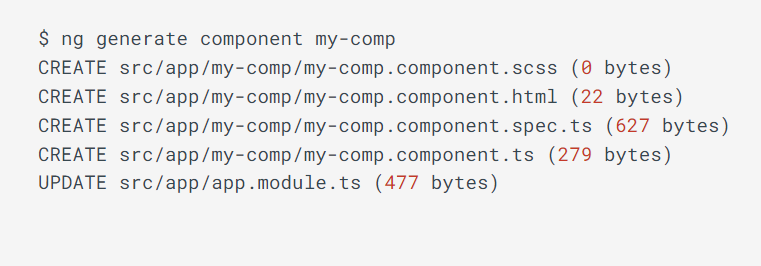
### 4.3. Компоненты: создание, основные хуки жизненного цикла

Angular в соответствии с современными стандартами фронтенд-разработки использует компонентно ориентированный подход в проектировании приложения. Поэтому компоненты являются самой важной сущностью в фреймворки.

Чтобы создать компонент, можно как вручную создать нужные файлы и сделать подключение к модулю, так и заскафолдить с помощью команды CLI:

$ ng generate component my-component

Результат выполнения данной команды:



Любой компонент состоит из двух обязательных вещей: внешнего вида (верстка и стили) и логика работы. На каждую часть создается отдельный файл, при этом стили выделяются в отдельный файл с настроенным нужным препроцессором стилей (либо просто css), бонусом по дефолту создается файл для тестов компонента.

Разберем полученную заготовку для нашего компонента:

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

@Component({

selector: 'app-my-comp',

templateUrl: './my-comp.component.html',

styleUrls: ['./my-comp.component.scss']

})

export class MyCompComponent implements OnInit {

constructor() { }

ngOnInit(): void {

}

}

Компонент является компонентом только с декоратором [Component](https://angular.io/api/core/Component)[[5]](#footnote-5), в котором перечислена основная метаинформация:

* селектор, с помощью которого можно вставлять данный компонент в другие компоненты в верстке
* пути до файлов с темплейтом (шаблоном) и стилями. В качестве альтернативы можно инлайнить шаблон и стили в файл с компонентом вместо указания путей, что может быть актуально когда компонент крошечный, и не хочется раздувать файловую структуру
* прочая метаинформация

Также мы видим пустой метод *ngOnInit*. Это один из хуков жизненного цикла компонента. Компоненты могут создаваться в рантайме, менять свое поведение и уничтожаться – именно это и есть жизненный цикл компонента. Всего есть есть [8 хуков](https://angular.io/guide/lifecycle-hooks)[[6]](#footnote-6):

1. *ngOnChanges* - этот метод срабатывает при изменении входных параметров данного компонента ([Input](https://angular.io/api/core/Input)[[7]](#footnote-7)), о чем поговорим далее. Если входным параметром является объект, то по умолчанию должна измениться ссылка на него (объект должен быть пересоздан). Очень важно, что метод будет вызван ТОЛЬКО при изменении инпутов, ничто более не может вызывать его. Может срабатывать многократно в зависимости от частоты изменения инпутов.
2. *ngOnInit* - начальная инициализация. Стоит отдавать предпочтение этому методу, а не конструктору, т.к. в конструкторе еще нет доступа ко входным параметрам, а еще неудобно писать тесты на компоненты. Вызывается единожды.
3. *ngDoCheck* - кастомизация логики определения изменений в инпутах. Если Angular не может или не будет что-то обнаруживать изменения (например, при мутации объекта переданного в инпуты), то в данном методе мы можем скорректировать проверку и выполнить дополнительную логику, если изменения все же есть. Вызывается очень часто, при каждом change detection, поэтому крайне не рекомендуется располагать логику тяжелых вычислений (например работа с DOM, подобные операции затратны) в данном хуке. Данный материал находится за пределами данного методического пособия.
4. *ngAfterContentInit* - срабатывает единожды после инстанцирования и полной инициализации вложенных компонентов переданных через [content projection](https://angular.io/guide/content-projection)[[8]](#footnote-8) (применяется при создании компонентов оберток над произвольным контентом). Начиная с этого хука доступна работа с DOM в контенте текущего компонента, но не в его шаблоне. Данный материал находится за пределами данного методического пособия.
5. *ngAfterContentChecked* - срабатывает после каждого change detection. Дает возможность увидеть изменения в шаблоне контента текущего компонента (в контенте переданном через content projection). Данный материал находится за пределами данного методического пособия.
6. *ngAfterViewInit* - срабатывает единожды после инициализации вью (темплейта) данного компонента. Начиная с этого хука доступна работа с DOM в текущем компоненте. Данный материал находится за пределами данного методического пособия.
7. *ngAfterViewChecked* - аналогично *ngAfterContentChecked*, но для вью текущего компонента. Данный материал находится за пределами данного методического пособия.
8. *ngOnDestroy* - срабатывает единожды перед уничтожением компонента с последующем удалением из DOM. Нужно для уничтожения подписок на события и прочей логики по предотвращению утечек памяти.

### 4.4. Синтаксис шаблонов, вкладывание одного компонента в другой, передача данных между родительским компонентом и дочерними

Компонент должен служить выводу какой-нибудь информации, ее визуализации, или добавлять интерактивности для пользователя, поэтому доработаем сгенерированный пустой компонент. Попробуем создать некий компонент обратного отсчета. Для начала модифицируем шаблон:

<span>{{ currentCount }}</span>

<button

(click)="restart()"

[disabled]="!!currentCount">

Перезапустить

</button>

Можно отметить необычную разметку: фигурные скобки с указанием переменной, круглые скобки с вызовом функции и квадратные скобки с передачей все той же переменной приведенной к булин – все это примеры связывания данных (*data binding*).

* {{ currentCount }} - фигурные скобки означают [интерполяцию](https://angular.io/guide/interpolation)[[9]](#footnote-9). Это некое выражение, значение которого можно вывести, тут можно делать простейшие арифметические операций, конкатенировать строки, брать свойства у объектов, пользовать тернарными выражениями. Но на ряду с этим запрещено мутировать что-либо при интерполяции, некоторые операторы и синтаксические конструкции JS недоступны из-за своих сайд эффектов. В нашем примере выводиться одно из полей класса-компонента.
* (click)="restart()" - круглые скобки означают обработку события. Выражение используемое при обработке события быть вычислено при срабатывания онного события, именно поэтому в нашем примере вызывается один из методов компонента. События могут быть как нативными (click, mouseover, doubleclick и тд), так и созданными искусственно. Пример реализации искусственного события будет показан далее. Когда наш счетчик обратного отсчета до считает до конца, он кинет родительскому компоненту событие завершения.
* [disabled]="!!currentCount" - квадратные скобки означают передачу данных внутрь компонента, директивы, либо просто связывание данных уже с самими атрибутами DOM-узла.

Все поля и методы классов должны быть публичными (использовать модификатор private или protected для них не получится).

Добавим немного верстки в компонент:

:host {

display: flex;

width: 200px;

}

Селектор host указывает на корень текущего компонента. В нашем примере в DOM будет вставлен узел app-my-comp, к которому и будет применен данное правило, а в него уже будет вставлен шаблон, что мы описали ранее.

Теперь взглянем на логику работы компонента, тут изменений гораздо больше:

@Component({

selector: 'app-my-comp',

templateUrl: './my-comp.component.html',

styleUrls: ['./my-comp.component.scss']

})

export class MyCompComponent implements OnInit, OnDestroy {

@Input() countDown: number = 10;

@Output() timeIsOver: EventEmitter<void> = new EventEmitter();

currentCount!: number;

private intervalId!: any;

ngOnInit(): void {

this.restart();

}

ngOnDestroy() {

this.stop();

}

restart() {

this.currentCount = this.countDown;

this.intervalId = setInterval(() => {

if (this.countDown === 1) {

this.stop();

this.timeIsOver.emit();

}

this.currentCount--;

}, 1000);

}

private stop() {

clearInterval(this.intervalId);

}

}

* Поле countDown для стартового количества секунд, по дефолту 10, но данное значение можно переопределить через родительский компонент, т.к. данное свойство задекорировано с помощью *Input*, что означает входящий параметр компонента
* Поле timeIsOver нужно для эмита кастомных событий наверх родительскому компоненту. Достаточно вызывать метод *emit*, как это сделано в методе *stop*
* Как мы видим в методе ngOnInit при создании компонента, создается интервал, который срабатывает каждую секунду. id интервала сохраняется для остановке и для очистки памяти при убийстве компонента
* Если мы наш компонент уничтожаем, то интервал также очищается.

Осталась последняя правка. Уберем из родительского компонента все сгенеренную в самом начале верстку, добавим лишь:

<app-my-comp

[countDown]="15"

(timeIsOver)="alert()">

</app-my-comp>

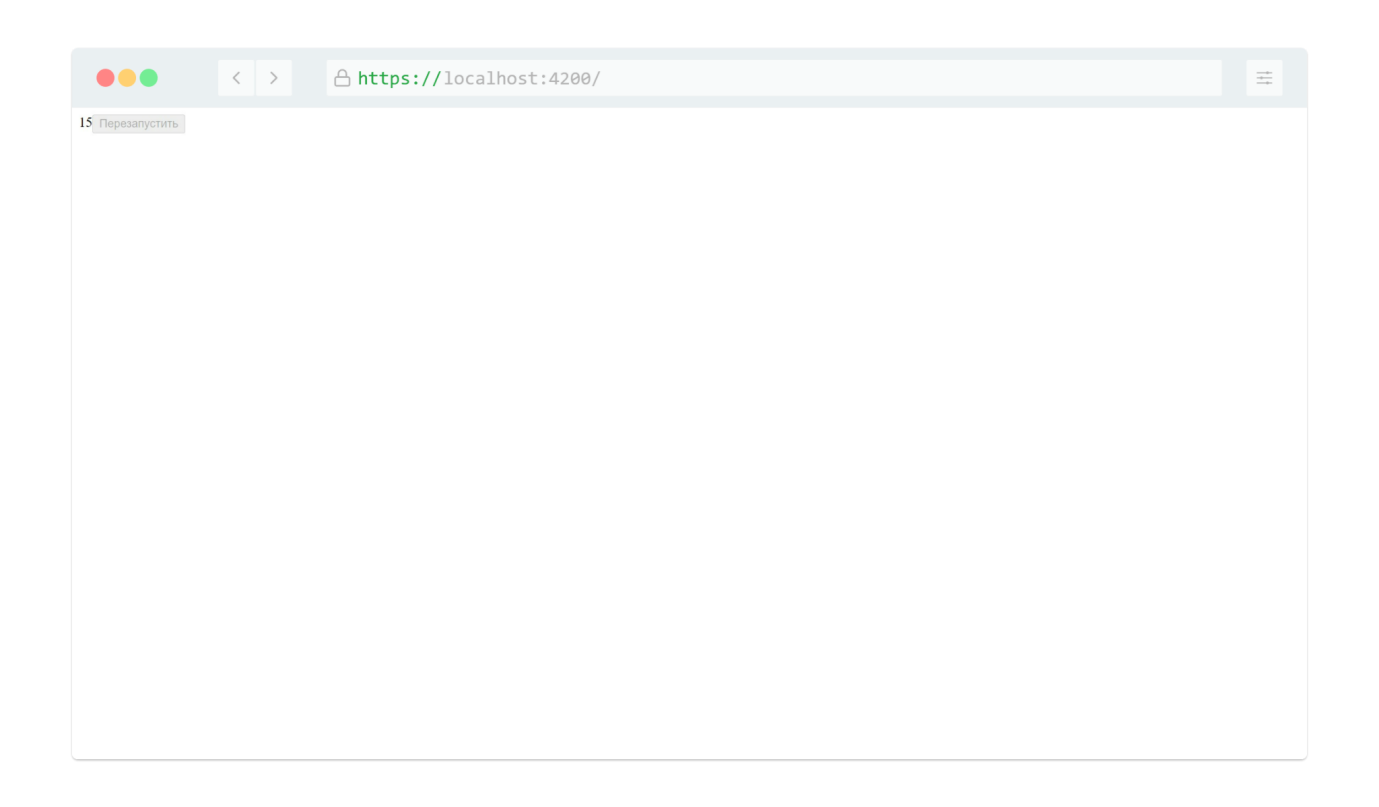
Как видно из примера, тут переопределено стартовое количество оставшихся секунд, а также назначен обработчик при истечении времени. В ts файле родителя пропишем метод обработки, в котором просто выводиться нативный *alert*:

alert() {

alert('Обратный отсчет закончен');

}

Запустив приложение, будет видно как тикает таймер, после чего выводиться сообщение о завершении. В конце также можно будет перезапустить таймер



**Упражнения для самостоятельной работы:**

Попробуйте создать свой собственный компонент, подключить его, вывести, реализовав логику общения родительского компонент и дочернего через Input и Output.

### 4.5. Директивы

[Директивы](https://angular.io/guide/built-in-directives#built-in-directives)[[10]](#footnote-10) является одной из механик инкапсуляции логики в рамках Angular, применяемой для соблюдения принципа единства ответственности. Если сравнивать их с компонентами, то у директив нет шаблона в отличии от последних. Верно и обратное утверждение: компоненты – это директивы с шаблоном. Таким образом, директивы служат для добавления функциональности для того или иного узла или компонента.

Стоит различать 2 вида директив:

1. [структурные](https://angular.io/guide/structural-directives)[[11]](#footnote-11). Нужны для изменения DOM, добавление и удаление узла с прикрепленной директивой в DOM. Например, директивы ngIf или ngFor.
2. [атрибутивные](https://angular.io/guide/attribute-directives)[[12]](#footnote-12). Нужны для добавление логики или стилей узлу DOM. Например, ngClass, ngStyle или ngModel.

Попробуем модифицировать предыдущие примеры:

<span [style.color]="currentCount ? 'black' : 'red'">

{{ currentCount }}

</span>

<button (click)="restart()" [disabled]="!!currentCount">

Перезапустить

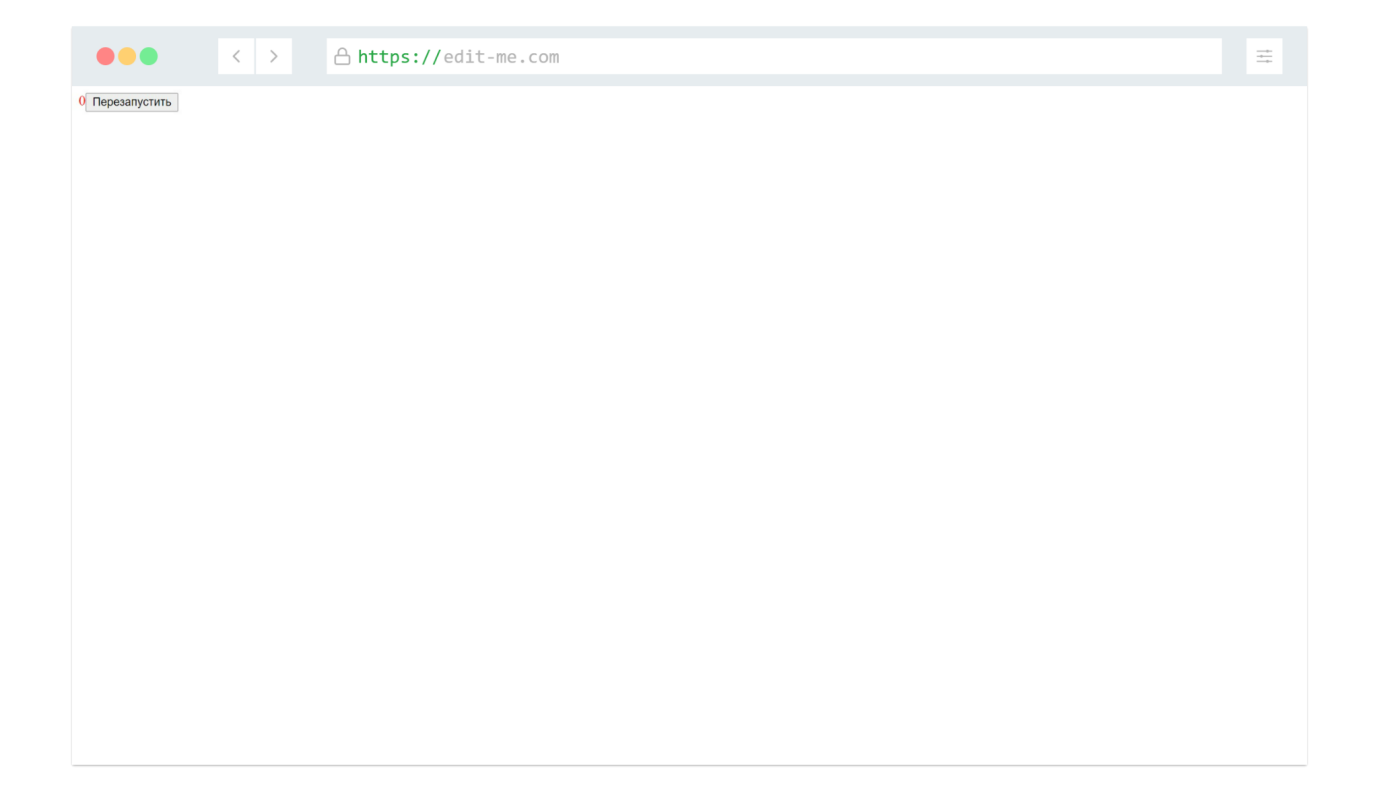
</button>

В данном случае мы применили директиву ngStyle, а именно ее сокращенный синтаксис для одного свойства. Полная запись выглядела бы

[ngStyle]="{color: currentCount ? 'black' : 'red'}"

Однако отметим, что при большом кол-ве свойств с большой вероятностью может подойти уже другая директива – *ngClass*, что позволит нам писать более чистые и поддерживаемый код.

Так что же делает директива в нашем пример? Как можно догадаться, она меняется свойство color (цвет текста) с черного на красный, когда количество оставшихся секунд станет falsy, т.е. достигнет 0, а из всех чисел только это приводится к false.



Попробуем внести еще одну модификацию в наш пример. Добавим в родительский компонент:

<app-my-comp

\*ngFor="let count of [5, 10, 15]"

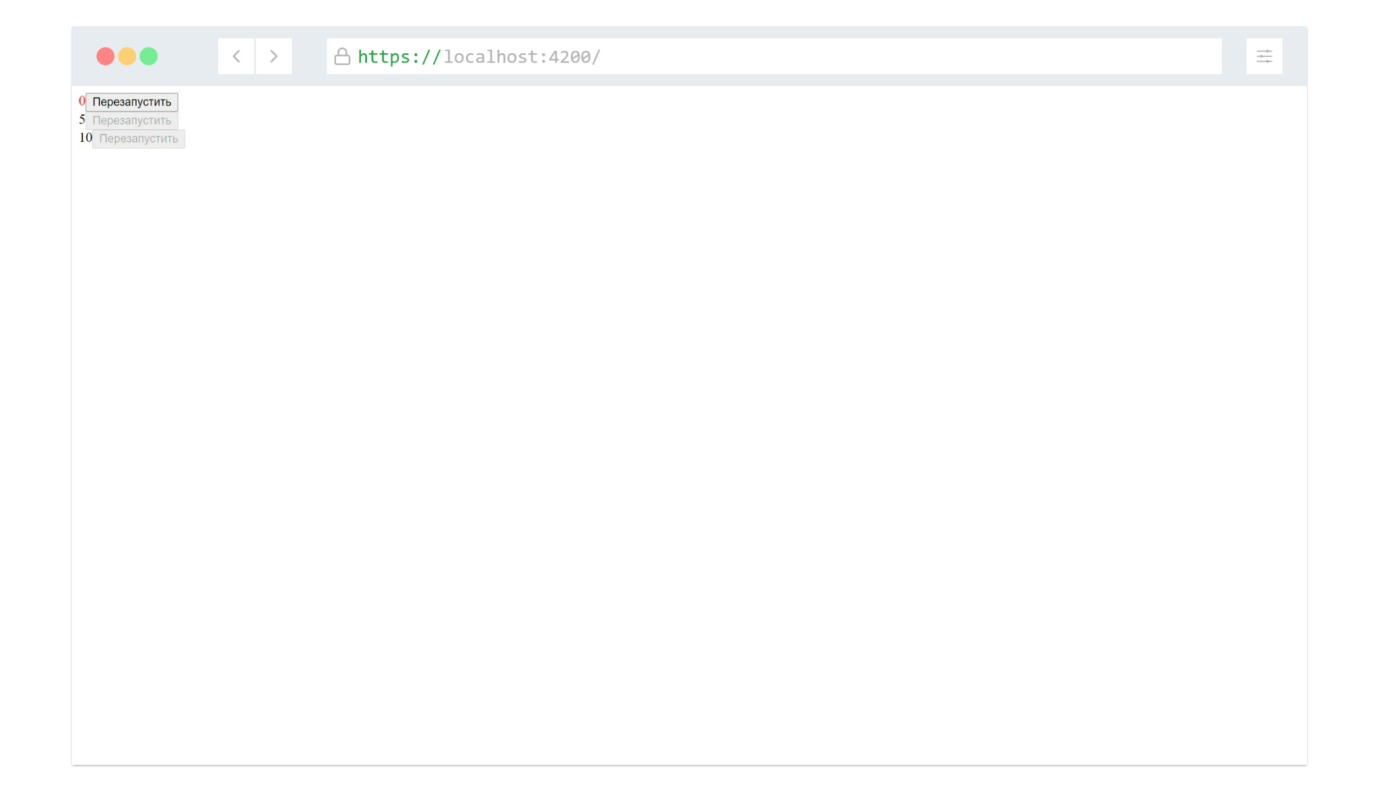
[countDown]="count"

(timeIsOver)="alert()">

</app-my-comp>

Тут уже добавлена директива ngFor для вывода фрагментов верстки циклом. Директива является структурной, на что указывает звездочка в специальном синтаксисе. Стоит отметить что “звездочка” это синтаксический синтаксис, благодаря которому данная конструкция заворачивается в служебный ангуляровский тег ng-template, с которым мы познакомимся позже.

Директива в нашем примере перебирает массив из трех элементов, создавая 3 таймера обратного отсчета, но с разным временем завершения:



**Упражнения для самостоятельной работы:**

Использование ngStyle зачастую бывает очень громоздким и неудобны – модифицируйте пример, перепишите его на [ngClass](https://angular.io/api/common/NgClass)[[13]](#footnote-13). Используйте также директиву [ngIf](https://angular.io/api/common/NgIf) [[14]](#footnote-14)для скрытия кнопки пока она не активна.

### 4.6. Сервисы

Как мы увидели общение между родительским и дочерним компонентом дело не трудное. Но что если у нас множество компонент вложенных друг в друга и нужно из самого вложенного передать данные в корневой или наоборот? Такая ситуация не редкость, компонентов может быть сколь угодно на страницы и вложены они могут быть тоже произвольно. В Angular данная проблема решена с помощью [сервисов](https://angular.io/guide/dependency-injection)[[15]](#footnote-15) (или иногда их называют провайдерами).

Как и компонент сервисы можно создать самому, а можно вызывать специальную команду CLI:

$ ng g service example-service

После чего будет сгенерирован файл:

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

@Injectable({

providedIn: 'root'

})

export class ExampleServiceService {

constructor() { }

}

Открыв файл мы увидим пустой класс, к которому применен декоратор Injectable. Работа с сервисами в Angular основана на применении паттерна Dependency Injection (DI), что служит уменьшению связанности кода и облегчению тестированию (DI позволяет с легкостью подкладывать заглушки вместе разных зависимостей во время тестов). Нам не нужно самим создавать экземпляры этого класса каждый раз когда нам нужен этот функционал – Angular инстанцирует их за нас и при повторном запросе инстанса вернет уже существующий, что позволит нам его переиспользовать, а также хранить в нем общие для нескольких компонент данные.

Как же выглядит запрос инстанса того или иного сервиса? Все просто. Достаточно указать его в конструкторе компонента или другого сервиса:

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

import { ExampleServiceService } from './example-service.service';

@Component({

selector: 'app-root',

templateUrl: './app.component.html',

styleUrls: ['./app.component.scss']

})

export class AppComponent {

constructor(private service: ExampleServiceService) {}

alert() {

// alert('Обратный отсчет закончен');

}

}

В рантайме при первом создании компонента будет инстанцирован данный сервис. Может показаться непонятным работа данного фрагмента кода, но тут сделано все для уменьшения действий разработчика при написании кода. Typescript прикрепляет метаинформацию о типе каждого аргумента к классу, Angular использует ее: тут достаточно знать ссылку на класс и порядок аргументов в конструкторе, чтобы корректно передать все зависимости, а тайпскрипт как раз и внедряет эти метаданные на этапе компиляции в js.

**Упражнения для самостоятельной работы:**

Перепишите компоненты с обратным отсчетом, чтобы не важно где они были и когда запускались, но таймеры у них были синхронизированы (отображалось единое время, кастомизаци времени отсчета утащить в сервис из инпутов, перезапуск один на всех действует).

### 4.7. Роутинг

Разберем теперь логику, без которой не обходится ни одно SPA приложение – маршрутизация на клиенте ([роутинг](https://angular.io/guide/routing-overview)[[16]](#footnote-16)). Для начала подключим [RouterModule](https://angular.io/api/router/RouterModule)[[17]](#footnote-17) и передадим ему пустой массив для конфигурации

const routes: Routes = [];

@NgModule({

declarations: [

AppComponent,

MyCompComponent

],

imports: [

BrowserModule,

RouterModule.forRoot(routes)

],

providers: [],

bootstrap: [AppComponent]

})

export class AppModule { }

Отметим необычный импорт модуля. RouterModule можно импортировать как есть, а можно результат вызова его статичных методов forRoot и forChild:

* RouterModule. Добавляет специальных директив для работы с навигацией приложения (например, routerLink)
* RouterModule.forRoot. Такой импорт должен быть только раз на все приложение. Тут содержится верхнеуровневая конфигурация нашей маршрутизации. Содержит директивы, как и предыдущий вариант, но также регистрирует сервисы [Router](https://angular.io/api/router/Router)[[18]](#footnote-18), [ActivatedRoute](https://angular.io/api/router/ActivatedRoute)[[19]](#footnote-19) и другие для получения информации по текущему роуту.
* RouterModule.forChild. Конфигурация дочернего модуля с ленивой подгрузкой. Данный материал находится за пределами данного методического пособия.

Попробуем теперь добавить новые страницы в наше приложение:

$ ng g c pageA

$ ng g c pageB

$ ng g c not-found

После чего добавим созданные компоненты в конфиг роутера:

const routes: Routes = [

{

path: 'a',

component: PageAComponent

},

{

path: 'b',

component: PageBComponent

},

{

path: '\*\*',

component: NotFoundComponent

}

];

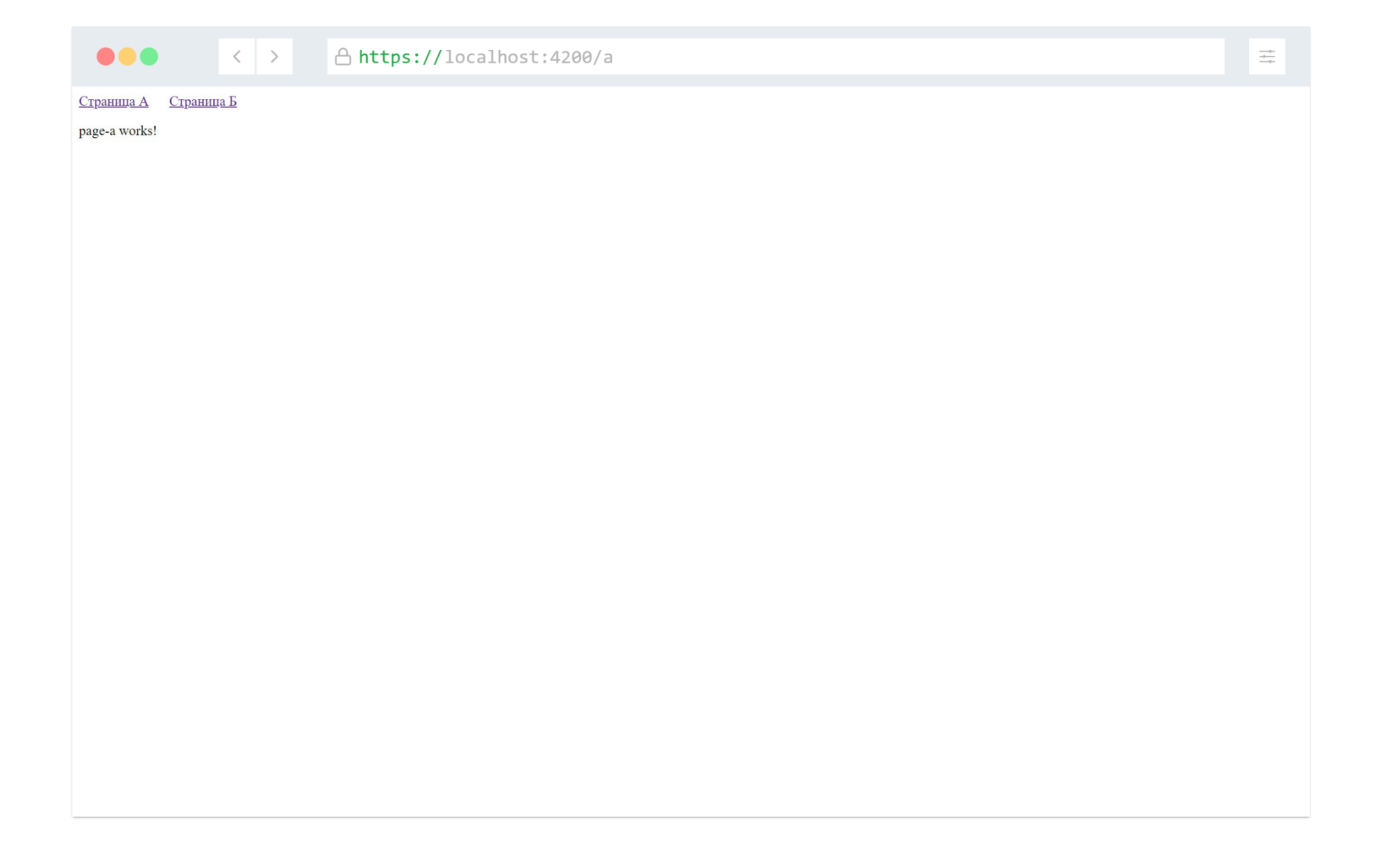
Далее добавим router-outlet. Это место в верстке, куда будет выводиться компоненты-страницы:

<a routerLink="/a">Страница А</a>

<a routerLink="/b" style="margin-left: 24px">Страница Б</a>

<router-outlet></router-outlet>

Теперь при запуске приложения мы всегда будем видеть две ссылки на верху страницы, контент же будет меняться в зависимости от того по какой ссылке кликнули. Также переход можно осуществить введя урл явно в адресную строку браузера. На случай ввода неизвестного урла (не /a и не /b в нашем случае) будет выведена страница-заглушка. Иногда эту страницу еще называют “Страница 404”.



## Глава 4. Основы фреймворка Vue.JS

### 4.1. Создание и настройка готового простого SPA приложения

Один из лучших вариантов для создание проектов на Vue.js - это инструмент командной строки Vue CLI.

Vue CLI - это полноценная система для быстрой разработки на Vue.js. Более подробную информацию можно увидеть на [официальном сайте этого проекта](https://cli.vuejs.org/ru/guide/).

Для начала следует установить глобальную версию этой команды чтобы, в дальнейшем, можно было легко и быстро создавать проекты в любой папке:  
npm install -g @vue/cli

После установки у вас появится доступ к команде vue в командной строке. Для создания нового проекта запустите команду:

vue create project-name

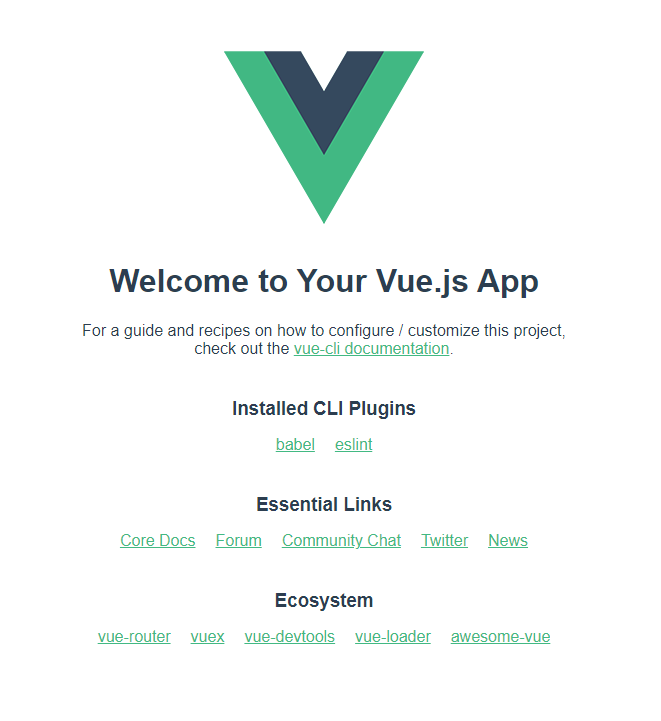
где project-name - это название проекта.  
При запуске этой команды можно выбрать как создавать проект. Например, проект на Vue3 или в ручном режиме с выбором всех нужных настроек. Более подробный процесс создания проекта с помощью Vue CLI можно увидеть на [официальном сайте](https://cli.vuejs.org/ru/guide/creating-a-project.html#vue-create).

В результате выполнения команды будет создана папка с проектом. Для запуска проекта следует зайти в папку с проектом:  
cd project-name

И запустить команду, которая запустит проект:

npm run serve

По-умолчанию, для проекта будет использоваться порт 8080. Если он будет занят, то проект запустится на другом порту, который будет указан после запуска проекта. Также, после запуска проекта, в командной строке будет ссылка на сам проект. Как правило, это <http://localhost:8080>. Пройдя по этой ссылке, в браузере откроется проект:



Теперь можно изменять компоненты приложения и, сразу же, видеть изменения на странице в браузере.

### 4.2. Описание структуры шаблона приложения

Возьмем за пример проект, который позволяет получить информацию о футбольных лигах и командах. Структура папок и файлов получится примерно такая:

public

├── [favicon.ico](https://github.com/ksuvers/vue-simbirsoft/blob/main/public/favicon.ico)

├── [index.html](https://github.com/ksuvers/vue-simbirsoft/blob/main/public/index.html)

src

├── api

│ ├── axios.js

│ ├── leagues.js

│ └── teams.js

├── assets

├── components

│ ├── Header.vue

│ └── Loader.vue

├── router

│ ├── index.js

├── store

│ ├── index.js

│ ├── league-calendar.js

│ ├── leagues-list.js

│ ├── team-calendar.js

│ ├── teams-list.js

├── views

│ ├── Home.vue

│ ├── LeagueCalendar.vue

│ ├── LeaguesList.vue

│ ├── TeamCalendar.vue

│ ├── TeamsList.vue

├── App.vue

└── main.js

.browserslistrc

.env

.eslintrc.js

.gitignore

README.md

package-lock.json

package.json

vue.config.js

Описание структуры файлов и папок:

* public – это папка с основным html-файлом [index.html](https://github.com/ksuvers/vue-simbirsoft/blob/main/public/index.html), который будет загружаться на странице браузера и в котором подключено приложение. Также в этой папке хранится фавиконка проекта.
* src – папка, в которой хранятся все основные js-файлы и компоненты vue приложения.
  + api – папка с js-файлами, отвечающими за API проекта. В данном случае это основной файл библиотеки axios и два api-файла для получения данных о лигах и командах.
  + assets – в этой папке, как правило, хранятся svg-иконки, css-файлы, изображения (png, jpg, …), шрифты и прочие вспомогательные файлы проекта.
  + components – основная папка с компонентами проекта.
  + router – папка с роутером. Как правило обходятся одним файлом index.js в котором описывают все необходимые пути страниц.
  + store – папка с состоянием проекта (vuex). Основной файл index.js и файлы для работы с логическими модулями проекта. Также, можно встретить реализации с разбиением файла index.js на действия и мутации.
  + views – папка с заглавными компонентами (так называемыми страницами). Еще эту папку называют pages. В ней хранятся компоненты-страницы, на которые указывает роутер. Т.е. это те компоненты, которые будут открываться при открытии определенных страниц и, уже внутри себя, подключать другие компоненты.
  + App.vue – основной vue-компонент проекта, который загружается в первую очередь, а уже в нем подключаются все остальные компоненты. Можно использовать в качестве шаблона.
  + main.js – основной js-файл проекта, в котором можно подключать плагины, библиотеки, а также выполнять прочие действия, связанные с инициализацией приложения.
* .browserslistrc – файл, в котором можно описать целевые браузеры приложения. Т.е. браузеры, под которые будет собрано приложение.
* .env – файл, в котором можно указать переменные окружения для сборки проекта на сервере. Например, токен google-карт.
* .eslintrc.js – в этом файле описываются конфигурация и настройки инструмента eslint, который помогает писать красивый и правильный код, указывая на ошибки.
* .gitignore – файл, в котором указывается список файлов и папок, которые не нужно отслеживать гитом (например, node\_modules, папки с конфигами ide, папка с билдом при локальной сборке).
* README.md – файл с описание проекта.
* package-lock.json – в этом файле сохраняются версии установленных пакетов. Актуально, когда зависимости заданы в виде интервалов версий, а нужно чтобы у всех в команде были одинаковые версии.
* package.json – список зависимостей проекта, а также список настроенных скриптов.
* vue.config.js – конфигурационный файл vue, в котором можно задать разные настройки сборки проекта: пути и названия файлов, настройки webpack и прочее.

### 4.3. Экземпляр Vue, основные хуки

Экземпляр компонента Vue - это объект класса Vue, который в качестве параметра принимает опции:

var vm = new Vue({

// опции

})

Более подробно с экземпляром компонента Vue можно ознакомится [здесь](https://ru.vuejs.org/v2/guide/instance.html).

Опции, в свою очередь, являются объектом с предопределенными полями.

Продолжая говорить о нашем проекте о лигах и командах, реализуем компонент календаря лиг. Для этого нам понадобятся почти все основные опции экземпляра компонента:

{

name: "LeagueCalendar",

components: {

AppLoader

},

props: {

id: {

type: Number,

required: true

},

year: {

type: Number

}

},

data() {

return {

rangeFilter: null

};

},

computed: {

...mapGetters({

allMatches: getterTypes.getAllMatches,

isLoading: getterTypes.isLoading

}),

yearFilter: {

get() {

return this.year || "";

},

set(value) {

if (Number.isInteger(+value)) {

this.$router

.replace({

name: this.$route.name,

query: { ...this.$route.query, year: value }

})

.catch(() => {});

}

}

},

preparedMatches() {

const matches = [...this.allMatches];

return matches.map(match => {

match.localDate = new Date(match.utcDate).toLocaleString();

return match;

});

},

filteredMatches() {

let matches = [...this.preparedMatches];

// фильтрация по диапазону дат

if (this.rangeFilter) {

matches = matches.filter(match => {

const matchDateInstance = new Date(match.utcDate);

const startDateInstance = this.rangeFilter[0];

const endDateInstance = this.rangeFilter[1];

return (

matchDateInstance >= startDateInstance &&

matchDateInstance <= endDateInstance

);

});

}

// фильтрация по году

// не производится, если стоит фильтр по диапазону дат

if (this.yearFilter && !this.rangeFilter) {

matches = matches.filter(match => {

const yearOfMatch = new Date(match.utcDate).getFullYear();

return String(yearOfMatch).startsWith(String(this.yearFilter));

});

}

return matches;

}

},

watch: {

yearFilter(newValue) {

this.rangeFilter = null; // если выбирается фильтр по дате, то сбрасывается фильтр по диапазону дат

if (newValue === "") {

// если новое значение фильтра пустое, то убираем этот фильтр из get параметров

this.deleteQueryFromRoute("year");

}

}

},

methods: {

deleteQueryFromRoute(queryName) {

const query = { ...this.$route.query };

delete query[queryName];

this.$router.replace({ name: this.$route.name, query: { ...query } });

}

},

created() {

this.$store.dispatch(actionTypes.getLeagueCalendar, {

id: this.$route.params.id

});

}

}

Описание опций компонента Vue из этого примера:

* name – Имя компонента. Можно не задавать. Как правило, используется для отладки – сообщениях консоли и во vue dev tools будут отображаться указанными именами, а не назначенными автоматически.
* components – Используется для перечисления компонентов, которые используются в текущем. В нашем случае, это компонент AppLoader, который нужен для визуализации загрузки.
* props – Параметры компонента. С помощью этой опции можно принимать переданные в компонент параметры, заданные тэгами.
* data – Объект с данными компонента. Все определенные в этой опции данные будут не только доступны в шаблоне компонента, но и станут реактивными, т.е. при их изменении будет меняться и шаблон.
* computed – Определение вычисляемых свойств. С этой опцией мы ознакомимся позже.
* watch – Используется для слежения за свойствами компонента. С этой опцией мы ознакомимся позже.
* methods – Методы. Являются функциями, которые можно вызывать и друг из друга и в шаблоне. Как правило, используются для работы с данными компонента.
* created – Хук жизненного цикла. О хуках жизненного цикла в следующем абзаце.

Более подробно ознакомиться с опциями компонента можно [здесь](https://ru.vuejs.org/v2/api/#%D0%9E%D0%BF%D1%86%D0%B8%D0%B8-%E2%80%94-%D0%B4%D0%B0%D0%BD%D0%BD%D1%8B%D0%B5).

При создании и дальнейшей работе компонента, он проходит множество шагов, между которыми вызываются определенные функции, которые называются хуками жизненного цикла. Вот основные из них:

* created – этот хук выполняется после создания компонента, но перед отрисовкой шаблона. Как правило, используется для первичного получения данных и прочих обработок, которые будут использоваться в шаблоне. Как можно увидеть, в нашем примере этот хук используется для получения данных о календаре лиг:

created() {

this.$store.dispatch(actionTypes.getLeagueCalendar, {

id: this.$route.params.id

});

}

* mounted – этот хук выполняется после отрисовки шаблона. Как правило этот хук может быть полезен для первичной обработки данных, связанных с шаблоном. Например, для добавления событий элементам шаблона.
* destroyed – этот хук выполняется после уничтожения компонента. Как правило, используется для удаления данных или событий, которые не будут нужны после уничтожения компонента. Например, в этом хуке можно удалить ранее назначенного слушателя события, чтобы он не продолжал слушать событие после уничтожения компонента.

Более подробно ознакомиться с хуками жизненного цикла компонента можно [здесь](https://ru.vuejs.org/v2/guide/instance.html#%D0%A5%D1%83%D0%BA%D0%B8-%D0%B6%D0%B8%D0%B7%D0%BD%D0%B5%D0%BD%D0%BD%D0%BE%D0%B3%D0%BE-%D1%86%D0%B8%D0%BA%D0%BB%D0%B0-%D1%8D%D0%BA%D0%B7%D0%B5%D0%BC%D0%BF%D0%BB%D1%8F%D1%80%D0%B0).

Основа работы компонента Vue заключается в шаблоне, данных и их обработки и реактивностью между шаблоном и данными.

Рассмотрим пример взаимодействия данных компонента и его шаблона на примере нашего проекта. В шаблоне компонента для отображения списка лиг требуется отобразить колонку с названием места проведения только если страница имеет ширину более 700 пикселей. Для этого в данных компонента задается свойство windowWidth:

data() {

return {

windowWidth: 0

};

},

Далее создается метод для задания текущей ширины экрана:

methods: {

...

setWindowWidth() {

this.windowWidth = document.documentElement.clientWidth;

}

},

Далее, после создания компонента, выполняется этот метод для первичного задания текущей ширины экрана и создается слушатель события изменения размера страницы, который при каждом изменении размера страницы будет заново назначать переменной текущую ширину окна.

created() {

...

this.setWindowWidth();

window.addEventListener("resize", this.setWindowWidth);

}

И, наконец, в шаблоне

<el-table-column

v-if="windowWidth > 700"

prop="area.name"

label="Area"

/>

В этом примере шаблона используется директива v-if, о которой будет описано в следующих главах, сейчас следует обратить внимание лишь на то, что при каждом изменении переменной windowWidth в методе setWindowWidth(), шаблон будет реагировать на эти изменения и, при необходимости, перерисовываться.

**Упражнение 1**. Создайте компонент, который будет выводить текст “Лок'Тар огар!”.

**Упражнение 2**. Создайте компонент, который будет выводить текст “created” в хуке created и текст “mounted” в хуке mounted.

### 4.4. Синтаксис шаблонов, директивы

Шаблон компонента Vue – это HTML-код. Для “оживления” шаблона, то есть для того чтобы добавить и манипулировать данными компонента в шаблоне, используются синтаксис Mustache и директивы.

Более подробно с синтаксисом шаблонов Vue компонента можно ознакомиться [здесь](https://ru.vuejs.org/v2/guide/syntax.html#%D0%94%D0%B8%D1%80%D0%B5%D0%BA%D1%82%D0%B8%D0%B2%D1%8B).

С помощью синтаксиса Mustache (двойных фигурных скобок) можно вставлять данные из компонента в шаблон. Для примера, можно вывести текущую ширину окна (заданную в data) из примера выше:

**<div>{{ windowWidth }}</div>**

В результате на странице будет отображено число, отражающее текущую ширину окна. И, при ее изменении, шаблон будет перерисовывать эту часть кода и отображать текущее значение этой переменной.

Более подробно об этом методе вывода данных в шаблоне можно ознакомиться [здесь](https://ru.vuejs.org/v2/guide/syntax.html#%D0%98%D0%BD%D1%82%D0%B5%D1%80%D0%BF%D0%BE%D0%BB%D1%8F%D1%86%D0%B8%D0%B8).

Для более тонкого и гибкого управления данными в шаблоне, во Vue есть несколько директив. Директивы — это специальные атрибуты с префиксом v-.

Список основных директив шаблона компонента:

1. v-if. Для примера применения этой директивы можно вернуться к списку лиг, в котором требуется скрыть колонку с местом проведения если ширина окна станет менее 700 пикселей:

<el-table-column

v-if="windowWidth > 700"

prop="area.name"

label="Area"

/>

В данном случае, директива v-if будет реагировать на изменения переменной windowWidth и удалять или вставлять колонку в зависимости от указанного в ней условия.

1. v-for. Для примера применения этой директивы, можно вывести список названий лиг из нашего проекта на страницу. В соответствующем компоненте уже есть вычисляемое свойство, которое возвращает массив с лигами:

filteredLeagues() {

return this.allLeagues

.filter(league => {

return this.searchFilter

? league.name.toLowerCase().match(this.searchFilter.toLowerCase())

: true;

})

.filter(league => {

const fullYear = new Date(

league.currentSeason.startDate

).getFullYear();

return this.yearFilter !== ""

? String(fullYear).startsWith(String(this.yearFilter))

: true;

});

},

Остается лишь использовать это в директиве v-for в шаблоне:

<div v-for="league in filteredLeagues" :key="league.id">{{ league.name }}</div>

Обратите внимание, при использовании этой директивы, всегда следует добавлять уникальный ключ :key="league.id чтобы Vue было легче отрисовывать элементы списка.

Более подробно о директивах можно ознакомиться [здесь](https://ru.vuejs.org/v2/guide/syntax.html#%D0%94%D0%B8%D1%80%D0%B5%D0%BA%D1%82%D0%B8%D0%B2%D1%8B).

Еще один очень полезный способ связать шаблон и данные компонента - события и их обработчики. Обработчики событий в шаблоне компонента Vue описываются с помощью символа @ и работают с двумя типами событий: нативные события html-элементов и события компонента.

1. Нативные события html-элемента – это классические события javascript (click, input, blur, …). Главное отличие в том, что в таких обработчиках можно не только напрямую написать javascript-код, но и использовать данные и методы из компонента. Для примера, добавим в шаблон компонента, отображающего список лиг, кнопку, которая будет задавать ширину окна:

**<button @click="windowsWidth=500">**

**Изменить ширину окна**

**</button>**

Теперь, при нажатии на эту кнопку, ширина окна будет приравниваться к 500.

Как писалось ранее, в событии можно также использовать и методы, что делает шаблон маленьким и понятным. То есть весь код, которые надо выполнить при наступлении события, можно поместить в метод компонента, а при наступлении события лишь вызвать этот метод. Изменим ширину окна, как в предыдущем примере, но уже с методом:

<button @click="setWindowWidth()">

Изменить ширину окна

</button>

1. События компонента. Во Vue события могут порождаться не только API браузера, но и самими компонентами (подробнее об этом будет в следующих главах). Таким образом, если написать обработчик события не у html-элемента, а у Vue-компонента, то обработчик будет срабатывать только на событиях, созданных компонентом. Для примера, можно рассмотреть шаблон главного тэга таблицы списка команд:

<el-table

v-else

:data="filteredTeams"

style="width: 100%"

@cell-click="cellClickHandler"

>

Здесь el-table является не html-тэгом, а Vue-компонентом, который порождает событие cell-click. Соответственно, обработчик будет срабатывать только на это событие компонента.

Более подробно об обработчиках событий можно ознакомиться [здесь](https://ru.vuejs.org/v2/guide/events.html).

### 4.5. Вычисляемые свойства и отслеживание изменений

Как уже было сказано, в шаблонах компонента можно использовать как данные компонента, так и целые выражения. Но это не всегда может быть читабельно и удобно. Рассмотрим пример из нашего проекта. Для вывода календаря лиг необходимо добавить в список лиг дату в локальном формате:

[...this.allMatches].map(match => {

match.localDate = new Date(match.utcDate).toLocaleString();

return match;

});

Если использовать этот код в шаблоне, то шаблон станет плохо читаемым и поддерживать этот код будет сложно:

<div

**v-for="league in [...this.allMatches].map((match) => {**

**match.localDate = new Date(match.utcDate).toLocaleString();**

**return match;**

**})"**

**:key="league.id"**

>

**{{ league.name }} ({{ league.localDate }})**

</div>

Исходя из уже полученных знаний, можно предположить, что для такого случая отлично подойдет метод. Например, так:

methods: {

getPreparedMatches () {

const matches = [...this.allMatches];

return matches.map(match => {

match.localDate = new Date(match.utcDate).toLocaleString();

return match;

});

},

},

И далее использовать этот метод в шаблоне:

<div

**v-for="league in getPreparedMatches()"**

**:key="league.id"**

>

**{{ league.name }} ({{ league.localDate }})**

</div>

Это сработает, но если исходный массив с данными изменится, то шаблон не перерисуется. Для таких случаев во Vue есть вычисляемые свойства. Вычисляемое свойство - это свойство объекта Vue, которое следит за изменениями реактивных данных внутри тела функции и, при их изменении, заново отрисовывает шаблон:

computed: {

getPreparedMatches () {

const matches = [...this.allMatches];

return matches.map(match => {

match.localDate = new Date(match.utcDate).toLocaleString();

return match;

});

},

},

Теперь, при каждом изменении реактивных данных в этой функции, шаблон будет перерисовываться:

<div

**v-for="league in getPreparedMatches"**

**:key="league.id"**

>

**{{ league.name }} ({{ league.localDate }})**

</div>

По умолчанию вычисляемые свойства работают только на чтение, но при необходимости можно также указать и сеттер. Для примера, возьмем вычисляемое свойство yearFilter:

computed: {

yearFilter: {

get() {

return this.year || "";

},

set(value) {

if (Number.isInteger(+value)) {

this.$router

.replace({

name: this.$route.name,

query: { ...this.$route.query, year: value }

})

.catch(() => {});

}

}

},

},

Теперь, при изменении свойства yearFilter будет вызываться указанный сеттер.

У вычисляемых свойств есть одна особенность – они кэшируются. Т.е. при многократном вызове вычисляемого свойства в шаблоне или коде компонента, это свойство не будет каждый раз пересчитываться. Вычисляемой свойство пересчитывается только если изменилась хоть одна реактивная зависимость этого свойства.

Более подробно с вычисляемыми свойствами можно ознакомиться [здесь](https://ru.vuejs.org/v2/guide/computed.html#%D0%92%D1%8B%D1%87%D0%B8%D1%81%D0%BB%D1%8F%D0%B5%D0%BC%D1%8B%D0%B5-%D1%81%D0%B2%D0%BE%D0%B9%D1%81%D1%82%D0%B2%D0%B0).

Еще один способ следить за изменениями реактивных данных - это слежение. Vue позволяет определить функции, которые будут вызываться каждый раз при изменении указанных реактивных данных компонента. Для примера можно рассмотреть фильтр по году в календаре лиг нашего проекта. У нас уже есть вычисляемое свойство yearFilter:

computed: {

yearFilter: {

get() {

return this.year || "";

},

set(value) {

if (Number.isInteger(+value)) {

this.$router

.replace({

name: this.$route.name,

query: { ...this.$route.query, year: value }

})

.catch(() => {});

}

}

},

},

Оно используется в шаблоне в качестве значения поля input для определения года в интерфейсе:

<el-input

v-model="yearFilter"

placeholder="Year"

class="search-input"

></el-input>

При вводе пользователем года в это поле, будет меняться свойство yearFilter, при изменении которого требуется выполнить определенные действия. Для этого можно определить слежение за этим свойством:

watch: {

yearFilter(newValue) {

this.rangeFilter = null; // если выбирается фильтр по дате, то сбрасывается фильтр по диапазону дат

if (newValue === "") {

// если новое значение фильтра пустое, то убираем этот фильтр из get параметров

this.deleteQueryFromRoute("year");

}

}

},

Таким образом, при изменении свойства yearFilter будет выполняться указанная функция.

У слежения есть особенность – в случае с объектом, по-умолчанию слежение происходит только за полями первого уровня, а за вложенными объектами слежения не происходит. Чтобы слежение осуществлялось за всеми вложенными объектами, следует передавать deep: true:

watch: {

yearFilter: {

deep: true,

handler(newValue) {

this.rangeFilter = null; // если выбирается фильтр по дате, то сбрасывается фильтр по диапазону дат

if (newValue === "") {

// если новое значение фильтра пустое, то убираем этот фильтр из get параметров

this.deleteQueryFromRoute("year");

}

}

}

},

Более подробно со слежением во Vue можно ознакомиться [здесь](https://ru.vuejs.org/v2/guide/computed.html#%D0%92%D1%8B%D1%87%D0%B8%D1%81%D0%BB%D1%8F%D0%B5%D0%BC%D1%8B%D0%B5-%D1%81%D0%B2%D0%BE%D0%B9%D1%81%D1%82%D0%B2%D0%B0-%D0%B8-%D1%81%D0%BB%D0%B5%D0%B6%D0%B5%D0%BD%D0%B8%D0%B5).

### 4.6. Родительские и дочерние компоненты

Vue позволяет использовать компоненты внутри других компонентов. Это очень удобная возможность – с ее помощью можно избежать создания больших компонентов и дублирования компонентов/кода.

Для примера рассмотрим компонент, отображающий процесс загрузки данных (“лоадер”). Этот компонент используется в нескольких других компонентах, поэтому было бы очень удобно один и тот же компонент просто использовать в других. Существует несколько походов для того чтобы вставить один компонент в другой. Стандартный подход заключается в следующем:

1. Необходимо локально зарегистрировать компонент.:

import AppLoader from "@/components/Loader";

export default {

...

components: {

AppLoader

},

...

}

Подробнее о регистрации компонентов можно ознакомиться [здесь](https://ru.vuejs.org/v2/guide/components-registration.html).

1. Далее указать этот компонент в шаблоне:

<app-loader v-if="isLoading" />

Теперь когда переменная isLoading будет равна true, в шаблоне родительского компонента будет появляться компонент app-loader. Таким образом, можно разбить любое приложение на множество обособленных и не очень компонентов и использовать их друг в друге. Это позволит избежать нагромождения шаблонов и кода и облегчит поддержку приложения.

Подключать компоненты в компонентах может быть очень удобно для простых компонентов, но, зачастую, требуется еще и передать какие-то данные в дочерний компонент из родительского. Это можно сделать с помощью атрибутов в тегах компонентов. Например, если в примере выше понадобиться добавить размер “лоадера”, то можно передать этот размер так:

<app-loader v-if="isLoading" :size=”40” />

И далее, в самом компоненте принять эти данные с помощью props:

export default {

props: {

size: {

type: Number,

default: 30

}

},

...

}

Теперь size доступно как свойство экземпляра компонента this.size. Изменять это свойство внутри дочернего компонента не рекомендуется, так как это может привести к осложнению отладки кода.

Более подробно с передачей данных в компоненты можно ознакомиться [здесь](https://ru.vuejs.org/v2/guide/components.html#%D0%9F%D0%B5%D1%80%D0%B5%D0%B4%D0%B0%D1%87%D0%B0-%D0%B4%D0%B0%D0%BD%D0%BD%D1%8B%D1%85-%D0%B2-%D0%B4%D0%BE%D1%87%D0%B5%D1%80%D0%BD%D0%B8%D0%B5-%D0%BA%D0%BE%D0%BC%D0%BF%D0%BE%D0%BD%D0%B5%D0%BD%D1%82%D1%8B-%D1%87%D0%B5%D1%80%D0%B5%D0%B7-%D0%B2%D1%85%D0%BE%D0%B4%D0%BD%D1%8B%D0%B5-%D0%BF%D0%B0%D1%80%D0%B0%D0%BC%D0%B5%D1%82%D1%80%D1%8B) и [здесь](https://ru.vuejs.org/v2/guide/components-props.html).

Помимо передачи данных в дочерний компонент, также можно передавать события от дочернего компонента к родительскому. Это делается с помощью функции emit. Если в любом дочернем компоненте написать this.$emit(‘event-name’, eventData) (вариант для кода компонента) или @click=”$emit(‘event-name’, eventData)” (вариант для шаблона), то в родительском компоненте можно будет поймать и обработать это событие (об этом уже писалось в пункте 11.13.4):

<my-component

@event-name="eventHandler($event)"

/>

где $event - это данные события.

**Упражнение 1**. Создайте компонент, в котором будет выводиться текст “Vue 3” (изначально скрытый) и еще один внутри него, в котором будет кнопка “Start!”. Сделайте так, чтобы при нажатии на кнопку вложенного компонента эмитировалось событие “start”, а родительский компонент по этому событию отображал текст “Vue 3”

### 4.7. Модальные окна, формы, валидация форм

Одним из самых наглядных примеров удобства и гибкости использования компонентов может быть модальное окно. Модальное окно это инструмент, который может использоваться на множестве страниц проекта. Данные, которые в нем меняются - заголовок и текст. Для реализации модального окна в качестве переиспользуемого компонента следует создать сам компонент, который будет отрисовывать само окно, принимать текст заголовка и содержимого и вставлять его в шаблон.

Шаблон:

<template>

<div class="modal-mask">

<div class="modal-wrapper">

<div class="modal-container">

<div class="modal-title">{{ title }}</div>

<div class="modal-body">{{ text }}</div>

<div class="modal-footer">

<button class="modal-button" @click="$emit('close')">

OK

</button>

</div>

</div>

</div>

</div>

</template>

Код компонента:

export default {

name: 'Modal',

props: {

title: {

type: String,

required: true,

},

text: {

type: String,

required: true,

}

}

}

Теперь в родительском компоненте достаточно лишь использовать этот компонент:

<template>

<modal @close="close()" title="Modal Title" text="Modal Text" />

</template>

export default {

...

methods: {

close () {

// Код, скрывающий модальное окно

}

}

}

Таким образом, компонент модального окна можно использовать в любом месте проекта, просто задавая заголовок и текст.

Формы.

Для работы с формами во Vue есть замечательная директива, которая используется для множества целей – v-model. С помощью этой директивы можно организовать двунаправленный обмен данными между шаблоном и данными компонента. Это очень удобно в разрезе форм. Рассмотрим на примере тега input. Как известно, для установки текста используется атрибут value, а для отдачи введенного текста используется событие @input. Директива v-model берет на себя обе задачи. На примере поля поиска в списке лиг нашего проекта:

<el-input

placeholder="Search"

v-model="searchFilter"

class="search-input"

></el-input>

Здесь, для связки данных, используется свойство searchFilter. Директива v-model делает так, что значение свойства searchFilter вводит текст с помощью value, а при вводе текста записывает текст из события input в свойство searchFilter. Что примечательно, эта директива для разных элементов форм использует разные, индивидуальные, свойства и события. Например, для элемента checkbox используются свойство checked и событие change.

Подробнее о работе с формами можно узнать [здесь](https://ru.vuejs.org/v2/guide/forms.html).

**Валидация форм**

Валидацию форм с помощью Vue можно выполнять несколькими способами. Как правило, для этого подключают специальные библиотеки, которые имеют обширный функционал для валидации и делают это удобно для разработчика. Рассмотрим простой способ без использования библиотек, только с помощью функционала Vue.

Допустим, на форме есть одно поле ввода для имени и кнопка отправки формы. Для текста поля ввода будет использоваться свойство text, а для аккумулирования ошибок будет использоваться свойство errors:

<form

id="app"

@submit="checkForm()"

action="https://vuejs.org/"

method="post"

>

<p v-if="errors.length">

<b>Пожалуйста исправьте указанные ошибки:</b>

<ul>

<li v-for="error in errors">{{ error }}</li>

</ul>

</p>

<p>

<label for="name">Имя</label>

<input

id="name"

v-model="name"

type="text"

name="name"

>

</p>

<p>

<input

type="submit"

value="Отправить"

>

</p>

</form>

Как видно из шаблона, при нажатии кнопки происходит вызов метода checkForm(). Остается только реализовать его:

export default {

data: {

errors: [],

name: null

},

methods: {

checkForm: function (event) {

if (this.name) {

return true;

}

this.errors = [];

if (!this.name) {

this.errors.push('Требуется указать имя.');

}

event.preventDefault();

}

}

}

Как видно из примера, метод проверки формы возвращает true если имя введено и отменяет отправку формы если имя не введено, попутно добавляя текст ошибки в массив с ошибками. Таким образом, если не ввести имя, то после нажатия кнопки на странице отобразится текст ошибки.

В методе проверки формы можно делать любые проверки. Например, проверить правильность введенной почты или произвести какие-то расчеты, на основе которых решить выводить ошибку или нет. В этом методе можно даже вызвать метод API сервера, на котором будет выполнена проверка формы.

Подробнее о работе с формами можно узнать [здесь](https://ru.vuejs.org/v2/cookbook/form-validation.html).

### 4.8. Использование и установка UI фреймворка Element

Из предыдущих примеров можно было убедиться как удобно делать компоненты и потом использовать их в разных местах проекта, просто передавая в них данные и получая от них события. Например, если в проекте много полей ввода, то можно создать один компонент с полем ввода и специфичной для проекта логикой и визуализацией и использовать его по всему проекту, не дублируя один и тот же код.

Если взглянуть на все современные веб-проекты более широким взглядом, то можно увидеть, что в большинстве из них используются одни и те же визуальные элементы – поля ввода, диалоги разных видов, меню, шаблоны и прочее. Чтобы не реализовывать все эти компоненты каждый раз в очередном проекте, можно воспользоваться готовыми решениями – на данный момент существует множество UI фреймворков/библиотек с целым набором готовых компонентов. Одной из таких библиотек является Element UI, которая используется в нашем проекте о футбольных лигах и командах.

Подключение большинства библиотек описано на сайтах этих библиотек. Рассмотрим как можно подключить Element UI. Для начала следует установить пакет с библиотекой:

npm i element-ui -S

Теперь, в файле main.js, следует подключить библиотеку к Vue:

import ElementUI from "element-ui";

import locale from "element-ui/lib/locale/lang/ru-RU";

Vue.use(ElementUI, { locale });

На этом этапе можно подключить еще много полезных вещей, таких как файлы css, темы и прочее. Теперь, когда библиотека подключена к Vue, можно просто писать имена компонентов этой библиотеки в шаблонах и на странице будут появляться готовые, красивые, функциональные и, что немаловажно, настраиваемые элементы. Рассмотрим несколько примеров использования компонентов библиотеки Element в нашем проекте.

Компонент выбора даты. Позволяет указывать тип элемента (в данном случае это диапазон дат - type="daterange"), выравнивание align="right", плейсхолдеры и многое другое:

<el-date-picker

v-model="rangeFilter"

type="daterange"

align="right"

unlink-panels

range-separator="To"

start-placeholder="Start date"

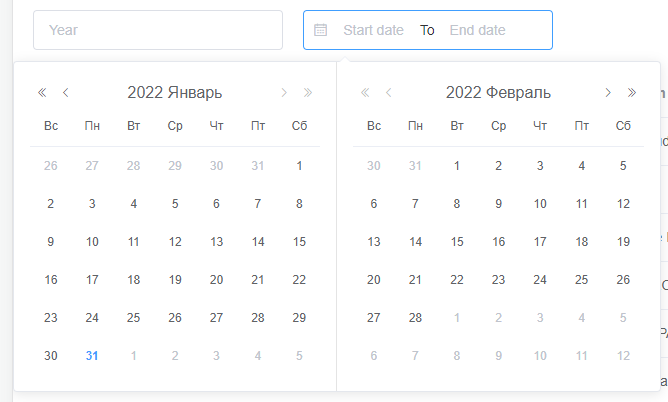
end-placeholder="End date"

class="search-input"

>

</el-date-picker>

Результат:



Компонент таблицы. Также имеет множество настроек:

<el-table :data="filteredMatches" style="width: 100%">

<el-table-column prop="homeTeam.name" label="Home Team" />

<el-table-column prop="awayTeam.name" label="Away Team" />

<el-table-column prop="localDate" label="Date" />

</el-table>

Результат:



Подобных библиотек множество. Например, знаменитый фреймворк Nuxt даже предлагает подключить подобную библиотеку из списка:

? UI framework: (Use arrow keys)

> None

Ant Design Vue

BalmUI

Bootstrap Vue

Buefy

Chakra UI

Element

Oruga

Primevue

Tachyons

Tailwind CSS

Windi CSS

Vant

View UI

Vuetify.js

**Упражнение 1**. Подключите библиотеку компонентов Vuetify и выведите в компоненте простой компонент библиотеки с текстовым полем v-text-field.

### 4.9. Плагины и фильтры

Плагины позволяют добавить во Vue некоторую глобальную функциональность: глобальный метод или свойство, свойство компонентов Vue или целую библиотеку. Область применения плагинов явно не определена, поэтому у них столь широкое применение. Например, описанная выше библиотека компонентов Component UI подключается именно как плагин:

Vue.use(ElementUI, { locale });

Также router тоже является плагином, который при подключении добавляет свойства $route и $router в компоненты Vue:

import VueRouter from "vue-router";

Vue.use(VueRouter);

Для создания плагина Vue необходимо создать объект с методом install. Этот метод будет вызываться с конструктором Vue в качестве первого аргумента, и с дополнительными опциями плагина в качестве второго (если передавались):

MyPlugin.install = function (Vue, options) {

Vue.prototype.$myMethod = function (methodOptions) {

// некоторая логика ...

}

}

Более подробно с плагинами можно ознакомиться [здесь](https://ru.vuejs.org/v2/guide/plugins.html).

Для повторяющихся задач форматирования текста отлично подходят фильтры. Они находят своё применение в двух местах: в mustache-интерполяциях и в выражениях v-bind. Для добавления фильтра достаточно в конце свойства Vue поставить вертикальную черту и название фильтра:

<div>{{ message | capitalize }}</div>

В данном случае, к свойству message применится фильтр capitalize перед отрисовкой элемента. Пример с выражением v-bind:

<div v-bind:id="rawId | formatId"></div>

Фильтр можно определить как в компоненте:

filters: {

capitalize: function (value) {

if (!value) return ''

value = value.toString()

return value.charAt(0).toUpperCase() + value.slice(1)

}

}

И тогда он будет доступен только в рамках компонента.

А вот добавление глобального фильтра:

Vue.filter('capitalize', function (value) {

if (!value) return ''

value = value.toString()

return value.charAt(0).toUpperCase() + value.slice(1)

})

В этом случае фильтр будет доступен в любом компоненте.

Также фильтры можно объединять в цепочки:

<div>{{ message | myFilter1 | myFilter2 }}</div>  
Подробнее с фильтрами можно познакомиться [здесь](https://ru.vuejs.org/v2/guide/filters.htm).

1. <https://unpkg.com/browse/@angular/core/package.json> – более детально список подерживаемых версии см в разделе engines [↑](#footnote-ref-1)
2. <https://angular.io/cli> раздел про CLI на оф сайте [↑](#footnote-ref-2)
3. <https://angular.io/cli/new> документация по команде ng new [↑](#footnote-ref-3)
4. <https://angular.io/cli/serve> документация по команде ng serve [↑](#footnote-ref-4)
5. <https://angular.io/api/core/Component> - документация по декоратору Component [↑](#footnote-ref-5)
6. <https://angular.io/guide/lifecycle-hooks> - хуки жизненного цикла компонента [↑](#footnote-ref-6)
7. <https://angular.io/api/core/Input> - документация по декоратору Input [↑](#footnote-ref-7)
8. <https://angular.io/guide/content-projection> - гайд по созданию компонентов оберток [↑](#footnote-ref-8)
9. <https://angular.io/guide/interpolation> - документация по возможностям и ограничениям при интерполяции [↑](#footnote-ref-9)
10. https://angular.io/guide/built-in-directives#built-in-directives [↑](#footnote-ref-10)
11. https://angular.io/guide/structural-directives [↑](#footnote-ref-11)
12. https://angular.io/guide/attribute-directives [↑](#footnote-ref-12)
13. https://angular.io/api/common/NgClass [↑](#footnote-ref-13)
14. https://angular.io/api/common/NgIf [↑](#footnote-ref-14)
15. https://angular.io/guide/dependency-injection [↑](#footnote-ref-15)
16. https://angular.io/guide/routing-overview [↑](#footnote-ref-16)
17. https://angular.io/api/router/RouterModule [↑](#footnote-ref-17)
18. https://angular.io/api/router/Router [↑](#footnote-ref-18)
19. https://angular.io/api/router/ActivatedRoute [↑](#footnote-ref-19)