# Основы React

## HotKeys

* Ctrl+F+K – автоформатирование
* Ctrl+Shift+F – поиск
* Ctrl+K+C – комментировать в JSX
* Alt+Ctrl – несколько курсоров плюс прыжки по строкам
* Ctrl+ Shift+Стрелка – выделить слово
* Написать div + Tab – быстрое создание тега

## Запуск первого React App

Делаем установку с [https://github.com/facebook/create-react-app через *node.js*](https://github.com/facebook/create-react-app%20через%20node.js) *command prompt.*

Переходим с помощью вышеупомянутой консольной утилиты в папку с будущим проектом и прописываем следующие вещи:

npx create-react-app my-app

cd my-app

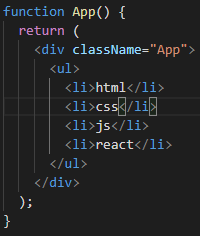
npm start

* npm install – загрузит папку node\_modules. Грузит модули на основании package.json
* \*npx в отличии от npm не только установит модули, но и подготовит их.
* \*вместо my-app можно свое название проекта
* package.json сообщает node.js, что папка является проектом в терминологии node.js

В файле App.js в функции App() должен быть только один корневой элемент

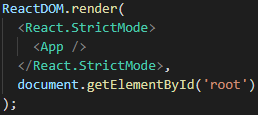
## Компонента

Компонента – это функция (если функция – то это stateless компонента), возвращающая разметку JSX. Пример:

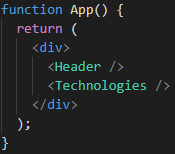
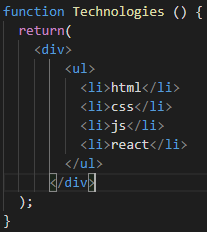
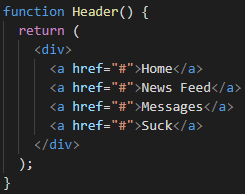


Соответственно появится тег <App/>. То есть также можно сказать, что компонента - это еще и тег.

Этот тег мы вставляем в файле *index.js*:



Одни такой тег можно вставить в другой тег, вот пример в App.js:

## Импорт/Экспорт

В реальном проекте каждая компонента (Technologies, Header), будет лежать в отдельном файле.

* ./ - относительно текущего местоположения
* ./.. – на уровень выше относительно текущего местоположения
* Импорт CSS отличается от импорта JS модулей: import './App.css';
* Модули из папки *node\_modules* импортируются по-другому:

import React from 'react'; - не нужно ставить никаких точек, нужная папка сама найдется

* Чтобы использовать синтаксис JSX нужно сделать импорт:

import React from 'react';

* Импорт картинки: import logo from './logo.png'; Вставка:

<img src={logo} alt='d' className="logo" />

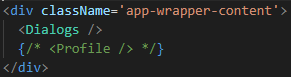
## Организация CSS

Для каждой компоненты нужно создать свой css файл. И подключить уже внутри каждой компоненты свой css файл.

* Чтобы css был изолирован в своих компонентах нужно назвать их так: name.module.css.
* Импорт такого css: import classes from './name.module.css'. classes – это объект, ключами которого являются названия стилей, а значения – сгенерированные для них автоматические названия:

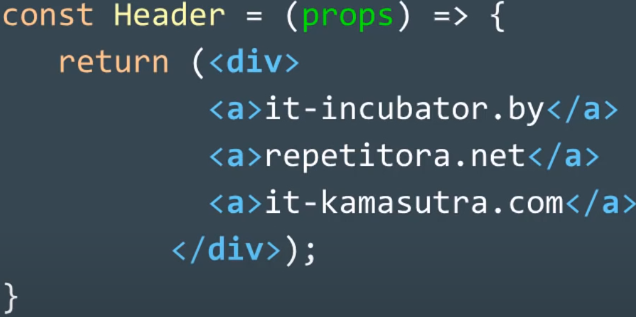


* Вот так применяются классы: <div className={classes.sideBar}>
* Два класса можно применить с помощью форматной строки: {`${s.class1} ${s.class2}`}
* Компонент и css группируем в одну папку для удобства
* Если у SPA приложения есть странички, которые меняются, но свойства контейнера у них одинаковые, их можно положить в тег div и вынести для него общие css свойства:



## Props

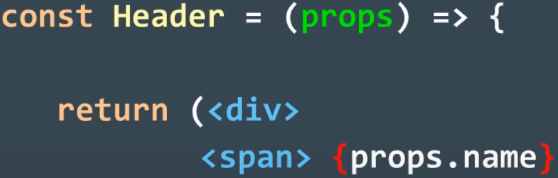
Пропсы – это параметр функции-компонента, которые содержит некоторые данные для нее:

 В качестве пропса всегда приходит какой-то объект. Когда вызываем так: <Header /> - то в пропсе будет пустой объект.

Чтобы передать объект с параметрами используются атрибуты компоненты:

  Каждый атрибут со значением становится парой ключ-значение объекта-параметра.

Использование:



**Все атрибуты тега-компоненты превращаются в Пробсы!**

## route, browser-router, маршрутизация

Система роутинга отслеживает адрес в адресной строке и будет реагировать на изменения в ней.

* Чтобы настроить эту систему нужно установить соответствующий модуль: npm install --save react-router-dom

save – внесет запись в package.json информацию об установленном нами модуле и после этого при *npm install* этот модуль будет автоматически загружаться в папку *node\_modules.*

### <Route/> - компонента

* Не забыть импортировать модули: import { BrowserRouter, Route } from 'react-router-dom';
* Все приложение (например внутри App.js) нужно обернуть в тег-компоненту <BrowserRouter >
* Пути настраиваются с помощью тега-компоненты <Route /> Пример:

Два способа:

1. <Route path='/dialogs' render={ () => <Dialogs />} />
2. <Route path='/profile' component={Profile} />

* Чтобы указать отрисовку компоненты, только в том случае, если url **точно** соответствует указанному, нужно добавить атрибут *exact*:

<Route ***exact*** path='/dialogs' component={Dialogs} /> То есть url вида */dialogs/something* теперь будет проигнорирован и отрисовка не произойдет.

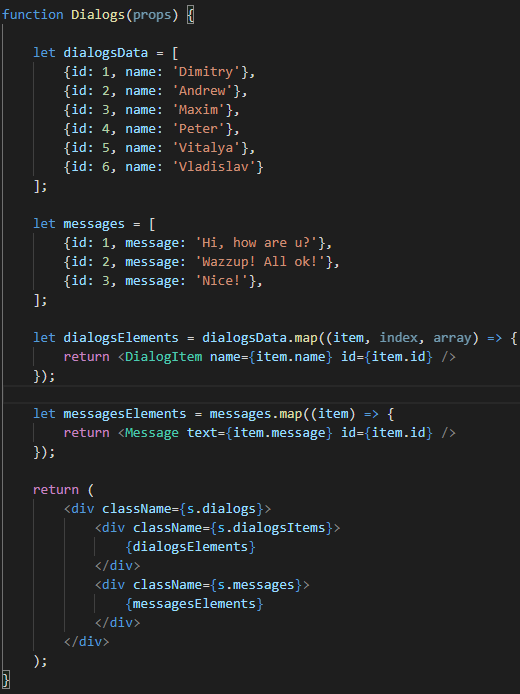
### <NavLink /> - компонента

* import { NavLink } from 'react-router-dom';
* <Navlink><Navlink /> используется вместо тега <a></a> чтобы менять URL и не перезагружать страницу снова, что по умолчанию делает <a>. Пример: <NavLink to='/profile'>Profile</NavLink>
* Атрибут activeClassName={obj.class} указывает какой класс применить для активной ссылки. По умолчанию ссылке ставится активный класс class=’active’

## UI - BLL, выносим данные в сторону

* BLL (Redux) – это некая data, выносим её отдельно от UI.
* UI – это компоненты (React).
* Данные в компоненту могут прийти в компонент через пробсы.

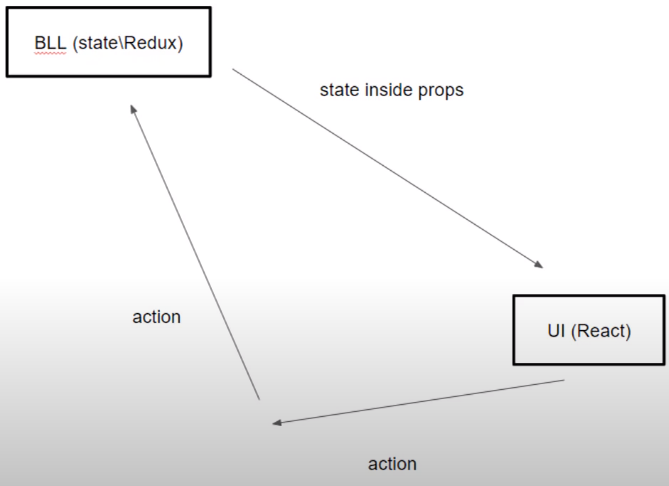
Если в JSX разметке находится массив компонентов, то эти компоненты вынуться из массива и отрисуются друг за другом. Массив компонентов обычно получают из массива объектов с помощью map. Пример:



### Прокидка через props

Все данные, такие как dialogsData выше, на самом деле должны приходить извне через пробсы. Если данные, которые пришли извне где-то изменились на уровне BLL, то UI должен перерисоваться. Также функции, которые делают изменения в BLL, тоже нужно прокинуть через props и вызвать в UI. При клике в UI – мы сначала делаем изменения в BLL, а только потом перерисовываем UI. Перерисовка UI при этом запускается из BLL

### FLUX-круговорот



Redux реализовывает FLUX архитектуру.

## onClick, ref, VirtualDOM

### Обработчики событий

* Перехватить событие можно так:



* Обращение к какому-либо html элементу из jsx разметки:

1. нужно дать элементу Реф (**ref**):



1. И теперь, мы можем создать ссылку на этот элемент: 
2. Свойство current ссылается на **нативный** HTML элемент (real DOM):



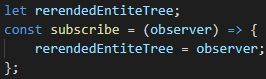
1. В React <textaria> пишется вот так:



Даже если попытаемся ввести символ, то свойство value мгновенно вернет текст, который был в этом свойстве, но в обработчике onChange мы можем отловить этот символ, до того, как он затрется.

## Паттерн Observer (callback, subscribe, observer)

В файле state.js определена переменная let rerendedEntiteTree. И функция subscribe:





В index.js мы импортируем subscribe и вызываем ее, передав ей в качестве аргумента некую функцию, которую хотим послать в мир state.js:





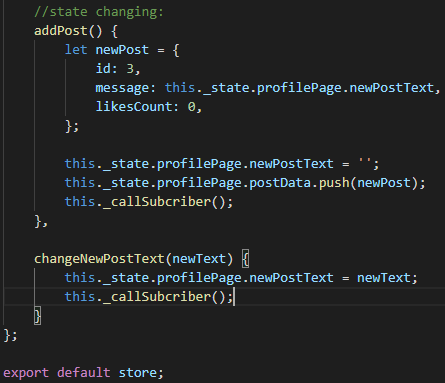
В итоге, в функции subscribe, переменная rerendedEntiteTree получает ссылку на функцию из файла index.js, и мы можем использовать ее в файле state.js

rerenderEntireTree – является подписчиком

### Использование паттерна

До внедрения dispatch и action «самодельный» Redux выглядит так:

* **state.js**



* **index.js:**

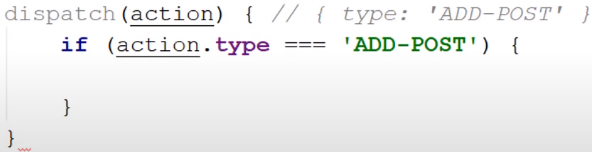


* **MyPosts.jsx** (как пример использования state.js):

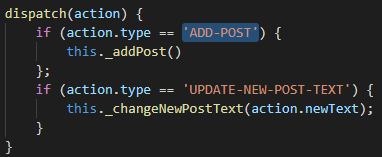


## Dispatch и action

Идея состоит в том, чтобы обращаться к объекту ***store*** через метод *dispatch* (который определим у store) и в качестве аргумента передавать объект action, на основании которого будет определяться то, что нужно сделать:

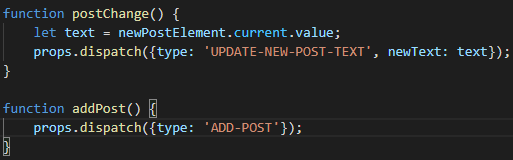


* **state.js** (dispatch)

****

Остальные методы в store остались те же.

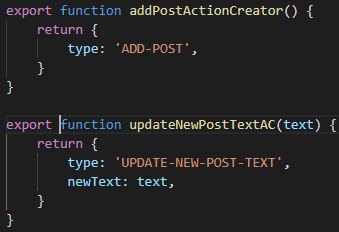
* **MyPosts.jsx** (как пример использования state.js)



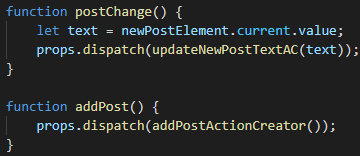
## Action creator, action type

В данной схеме у нас UI создает объект action самостоятельно. Это неудобно, т.к. можно ошибиться в названии свойства объекта, его количестве и т.д. Action creator() – это функция, которой мы делегируем создание action.

* **state.js**

****

* **MyPosts.jsx**

****

## Reducer (почти redux)

Редьюсеры позволяют распределить лишний код из метода dispatch.

**Reducer** – это функция, которая принимает state и action и возвращает новый state

В dispatch() вызываются редьюсеры, каждому передается своя часть стейта и объект action. Если в редьюсер пришел action с соответствующим типом события – он начинает работу по измене6нию своей часть стейта.