# Пререквизиты

Для начала работы на вашем компьютере должны быть:

1. Установлен Visual Studio Code. Скачать можно тут <https://code.visualstudio.com/>
2. Установлен Node.js. Скачиваем тут <https://nodejs.org/en/>
3. Установлен npm или yarn. Скачиваем тут <https://www.npmjs.com/>
4. Хорошее настроение, время, налитый горячий чай или кофе.

# Начало работы

1. Открываем проект в Visual Studio Code.
2. Запускаем npm install в терминале (yarn install)
3. В package.json есть скрипт start. Запускаем его (npm run start)
4. Открываем наш проект на порту 8080 – <http://localhost:8080>
5. Видим белый экран, радуемся

# Начальный HTML

Дальнейшая разработка будет происходить в src папке. Добавим статический код в src/App.jsx - вместо <div></div> вставим следующий код:

  <div className = "container">

    <header>

        <ul className="menu">

            <li><a href="#">Все новости</a></li>

              <li><a href="#">Создать</a></li>

        </ul>

    </header>

      <div className="top-container">

          <h1>Мы начинаем наш Воркшоп</h1>

          <div className="description">Для кого? Для чего?</div>

<div className = "green-line" />

      </div>

      <div className="news-container">

        <p>

            <span className="title">.NET Meetup #23</span>-<span className="dateTime">Sept 12 at 07:00PM </span>

        </p>

          <h3>Пришло время изучить что -то новенькое, и с этим нам помог Сергей Кривошеин, сделав обзор языка F#.</h3>

      <p>

      Держу пари, многие из вас, особенно кто пишет на C#, знают, что есть такой замечательный язык. На этом все знания о F#, возможно, заканчиваются. А вот Сергей по долгу службы и развития для использует этот язык. И в Карасике он поделился с нами, как же на нем писать, рассказал об основных синтаксических конструкциях, сравнил с C#, а еще поведал о всяких плюсах и минусах.

      </p>

      <p>

      И конечно же, сертификатик Сергею вручили – с ним все ж приятней всякие интересные штуки изучать.

        </p>

        <p>Смотрим бомбический видосик с попкорном и чайком!</p>

        <div className = "blue-line" />

      </div>

      <div className="news-container">

        <p>

            <span className="title">CoderGames </span>-<span className="dateTime">Sept 5 at 06:00PM</span>

        </p>

          <h3>Начало месяца ознаменовалось очередными CoderGames. </h3>

      <p>

          И на этот раз ребята сражались в знаменитую игру 2048. Всего участвовал 21 человек. Вот тут можно посмотреть фоточки, результаты и анонс. А пока поздравим победителей - Евгения Быкова, который занял первое место, а также Ольгу Газизову и Антона Канопкина.

      </p>

      <p>

        И конечно же, скажем спасибо организаторам и судьям за их поддержку. А всем участникам - молодцы!

        </p>

        <p>Кстати, тут недавно начался марафон CoderGames, который закончится в октябре. Мы вам о нем в следующий раз расскажем. </p>

        <div className = "green-line" />

      </div>

      <footer>

          <div>Парапарам</div>

          <img src ="https://bipbap.ru/wp-content/uploads/2017/12/65620375-6b2b57fa5c7189ba4e3841d592bd5fc1-800-640x426.jpg" />

      </footer>

  </div>

Чтобы выглядело все хорошо сначала вынесем часть с новостями в отдельный компонент Item.jsx.

Добавим в файл константу news = […] где … - это массив из файла db.json.

Затем вместо новостей будем использовать наш Item.jsx:

{ news.map((item, index) => <Item item={item} key={item.id} index={index} />)}

Создадим Footer.jsx.

Все, что лежит в App.jsx, вынесем в News.jsx – будет страница новостей. Визуально ничего не должно поменяться.

# Сделаем News.jsx умным компонентом и добавим ему state. Теперь у него есть свое собственное состояние, которое меняется внутри него.

import React, { Component } from 'react';

import Item from "./Item";

import Footer from "./Footer";

class News extends Component {

    constructor(props) {

        super(props);

        this.state = {

            news: news

        }

    }

    addNew () {

        let newList = [ ...this.state.news, news[0] ];

        this.setState({ news: newList });

    }

    render () {

        return (

<div className = "container">

        <header>

            <ul className="menu">

                <li>Все новости</li>

                <li>Создать</li>

                <li onClick={() => this.addNew()}>Добавить</li>

            </ul>

            <div>В статическом списке: {news.length}</div>

        </header>

        <div className="top-container">

            <h1>Мы начинаем наш Воркшоп</h1>

            <div className="description">Для кого? Для чего?</div>

            <div className = "green-line" />

        </div>

        {this.state.news.map((item, index) => <Item item={item} key={item.id} index={index} />)}

    <Footer />

  </div>);

    }

};

export default News;

# Жизненный цикл компонента.

Первоначальный рендеринг компонента в DOM называется «монтирование» (mounting). Каждый раз, когда DOM-узел, созданный компонентом, удаляется, происходит «размонтирование» (unmounting).

Чтобы сделать какие-то свои действия в это время есть две функции - componentDidMount, componentDidUnmount.

Метод componentDidMount() запускается после того, как компонент отрендерился в DOM.

В редких случаях может потребоваться позволить компоненту спрятать себя, ничего не показывать. Чтобы этого добиться, верните null. Значение вам вернуть надо обязательно, возврат null из метода render никак не влияет на срабатывание методов жизненного цикла компонента. Например, componentDidUpdate будет всё равно вызван.

componentDidUpdate вызывается, если props поменялись. При этом в качестве параметров в функцию приходит предыдущие значения пропсов. Метод срабатывает до того, как обновленное значение будет применено.

При монтировании компонента сделаем загрузку нужных нам данных.

  componentDidMount () {

        let result = fetch("http://localhost:4000/News")

        .then(response => response.json())

        .then(json => this.setState({ news: json, isLoading: false }));

    }

Чтобы показывать что-то, пока грузиться, сделаем Loader.jsx:

render () {

        if (this.state.isLoading) {

            return (<div className = "container">АААААААААААААААААААААААААААААААААААААААААААААА!!!!</div>)

        }

        return (<div className = "container">

Но лучше вынести в отдельный компонент – мы же будем его использовать частенько 😉

import React from 'react';

const Loader = () => (

    <div className = "container">АААААААААААААААААААААААААААААААААААААААААААААА!!!! ВСЕ ПРОПАЛО!!!!! Т\_Т</div>

);

export default Loader;

 if (this.state.isLoading) {

            return (<Loader />)

        }

Приведем нашу страничку в порядок. Выделим Container.jsx, в котором оставим Footer и Header.

React имеет мощную модель композиции, поэтому для повторного использования кода между компонентами мы рекомендуем использовать композицию вместо наследования. Некоторые компоненты не знают своих потомков заранее. Это особенно характерно для таких компонентов, как Sidebar или Dialog, которые представляют из себя как бы «коробку», в которую можно что-то положить.

Для таких компонентов мы рекомендуем использовать специальный проп children, который передаст дочерние элементы сразу на вывод.

const App = () => (

  <Container>

    <News />

  </Container>

);

class Container extends Component {

    constructor(props) {

        super(props);

    }

    render () {

      return (<div className = "container">

        <header>

            <ul className="menu">

                <li><a href="#">Все новости</a></li>

                <li><a href="#">Создать</a></li>

            </ul>

        </header>

        <div className="top-container">

            <h1>Мы начинаем наш Воркшоп</h1>

            <div className="description">Для кого? Для чего?</div>

            <div className = "green-line" />

        </div>

        {this.props.children}

    <Footer />

  </div>);

    }

};

А теперь добавим форму, которая позволит нам добавлять новую новость.

# Ключи

«Ключ» — это специальный строковый атрибут, который нужно указывать при создании списка элементов.

Ключи помогают React определять, какие элементы были изменены, добавлены или удалены. Их необходимо указывать, чтобы React мог сопоставлять элементы массива с течением времени: Лучший способ выбрать ключ — это использовать id, уникальный идентификатор. В крайнем случае можно использовать индекс элемента.

Ключи нужно определять непосредственно внутри массивов. Внутрь компонента ключ не попадает.

{

this.state.news.map((item, index) => <Item item={item} key={item.id} index={index} />)}

# Добавление Формы

Приведем нашу форму в порядок, выделив Container.jsx. Мы хотим добавить страницу с формой, роутера пока у нас нет, поэтому сделаем кривой роутинг.

import News from "./News";

import AddItem from "./AddItem";

class Container extends Component {

    constructor(props) {

        super(props);

        this.state = {

            currentPage: pages.list

        }

    }

    setPage (pageName) {

        this.setState({ currentPage: pageName });

    }

    render () {

      return (<div className = "container">

        <header>

            <ul className="menu">

                <li><a href="#" onClick= {() => this.setPage(pages.list)}>Все новости</a></li>

                <li><a href="#" onClick= {() => this.setPage(pages.add)}>Создать</a></li>

            </ul>

        </header>

        <div className="top-container">

            <h1>Мы начинаем наш Воркшоп</h1>

            <div className="description">Для кого? Для чего?</div>

            <div className = "green-line" />

        </div>

        { this.state.currentPage === pages.list && <News />}

        { this.state.currentPage === pages.add && <AddItem />}

    <Footer />

  </div>);

    }

};

const pages = { list: "list", add: "add" };

В зависимости от того куда тыкнули у нас будет либо показываться список, либо пустой элемент AddItem.jsx.

Давайте добавим нашу форму.

class AddItem extends Component {

    constructor(props) {

        super(props);

    }

    render () {

        return (<div className="form-container">

            <form>

                <div className = "row">

                    <label>Title:

                    <input type="text" name="title" />

                </label>

                </div>

                <div className = "row">

                <label>Description:

                    <input type="text" name="description" />

                </label>

                </div>

                <div className = "row">

                    <label>Info:

                    <input type="text" name="info" />

                </label>

                </div>

                <div className = "row">

                <label>Image(URL):

                    <input type="text" name="image" />

                </label>

                </div>

                <div className = "row">

                <button>Send</button>

                </div>

                </form>

        </div>)

    }

}

Форма работает так же, как и в обычном js – ничего не отправляет и редиректит дальше.

Давайте попробуем отправить ее в том виде, в котором она есть. И повесим на форму обработчик события handleSubmit.

handleSubmit (e) {

        let data = this.state.item;

        fetch("http://localhost:4000/News", {

            method: 'POST',

            headers: {

                'Content-Type': 'application/json'

            },

            body:  JSON.stringify(data)

        })

        .then (response => console.log(response.status))

        .catch(ex => {

            e.preventDefault()

        });

    }

Работать с формой можно двумя способами – через контролируемые объекты (рекомендуемая) и неконтролируемые (через ref)

Возьмем второй вариант.

Картинку можно взять <http://images6.fanpop.com/image/photos/34200000/Kawaii-kawaii-anime-34255208-320-375.jpg> отсюда или любую другую.

Для начала добавим обработчик события изменения инпута. Можно на каждое поле повесить свой собственный обработчик, но мы возьмем name нашего поля и value и для всех компонентов обработчик будет одинаковый:

 onChange (e) {

        let value = e.target.value;

         let name = e.target.name;

         this.setState(prevState => {

            return {

                item : { ...prevState.item, [name]: value }

            }

         });

    }

Предохранители

Предохранители это компоненты React, которые **отлавливают ошибки JavaScript в любом месте деревьев их дочерних компонентов, сохраняют их в журнале ошибок и выводят запасной UI** вместо рухнувшего дерева компонентов.

Напишем свой и поставим на самый верх.

export class ErrorBoundary extends React.Component {

    constructor(props) {

        super(props);

        this.state = { hasError: false };

      }

    componentDidCatch () {

        this.setState({ hasError: true });

    }

    static getDerivedStateFromError(error) {

        return { hasError: true };

      }

    render () {

        if (this.state.hasError) {

            return (

                <div>

                    <div>УУУУУпс.</div>

                </div>

            );

        }

        else

             return this.props.children;

    }

}

ErrorBoundary.propTypes = {

    children: PropTypes.any

};

export default ErrorBoundary;

Тестирование.

Для тестов мы будем использовать связку Jest + Enzyme. У нас будут модульные тесты, компоненты мы будем тестировать с помощью Enzyme в окружении Jest без полного рендеринга. Такие тесты быстрые и помогают проверить каждый компонент без вложенных.

Npm install –save-dev enzyme jest react-test-renderer enzyme-adapter-react-16

Для начала настроим Jest.

Добавим файл jest.config.js. Конфигурацию можно задать и в package.json, она должна быть в самом начале – но лучше пусть лежит отдельно.

Конфигурационный файл:

module.exports = {

    verbose: true,

    moduleFileExtensions: ['js', 'jsx'],

    setupFiles: ["./tests/setup.js"],

    transformIgnorePatterns: ['<rootDir>/node\_modules/'],

    transform: {

        '^.+\\.js[x]?$': 'babel-jest'

      },

};

Setup.js с настройками Enzyme:

require('@babel/register');

require('@babel/polyfill');

// Сделаем функции Enzyme доступными во всех файлах тестов без необходимости импорта importing

import Enzyme, { shallow, render, mount } from 'enzyme';

global.shallow = shallow;

global.render = render;

global.mount = mount;

// Обрушим тест при любой ошибке

console.error = message => {

   throw new Error(message);

};

import Adapter from "enzyme-adapter-react-16";

Enzyme.configure({ adapter: new Adapter() });

# Первый тест:

import React from "react";

import Loader from "../src/Loader";

describe("Test Loader component", () => {

    it("should show correctly", () => {

        const wrapper = shallow(<Loader  />);

        expect(wrapper.getElements().length).toBe(1);

        const text = wrapper.text();

        expect(text).toBe("АААААААААААААААААААААААААААААААААААААААААААААА!!!! ВСЕ ПРОПАЛО!!!!! Т\_Т");

    });

});

# Тестирование fetch и начального состояния:

 describe("Test News component", () => {

    it('fetches data from server when server returns a successful response', done => {

        const mockSuccessResponse = [{ Id: "1", Title: "Test", Info: "Test 2"}];

        const mockJsonPromise = Promise.resolve(mockSuccessResponse);

        const mockFetchPromise = Promise.resolve({

          json: () => mockJsonPromise,

        });

        global.fetch = jest.fn().mockImplementation(() => mockFetchPromise);

        const wrapper = shallow(<News />);

        expect(global.fetch).toHaveBeenCalledTimes(1);

        expect(global.fetch).toHaveBeenCalledWith('http://localhost:4000/News');

        process.nextTick(() => { // 6

          expect(wrapper.state()).toEqual({

            news: mockSuccessResponse,

            isLoading: false

          });

          global.fetch.mockClear();

        delete global.fetch;

        done();

        });

      });

});