React Complete Course Schwarzmuller

1. Getting started

2. JavaScript Refresher

**Classes**

Для чего нужен super() ?

Например, есть два класса, у обоих есть constructor. Он служит для того, чтобы у экземпляра появились **свойства** класса. **Методы** прописываются ниже в теле класса.

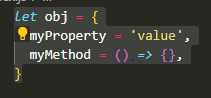
Если мы расширяем первый класс вторым, то, чтобы наследовался конструктор родительского класса, соответственно *чтобы появились* ***свойства*** *у экземпляра объекта от первого (наследуемого) класса, используем слово super().*

***Если кратко – используем super(), чтобы конструктор родительского класса передался дочернему.***

Именно поэтому в классовых реакт компонентах мы пишем каждый раз super у дочерних компонент.

**Classes, Properties, Methods**

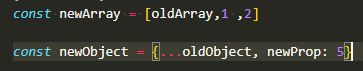
Современный синтаксис по написанию свойств и методов объектов:



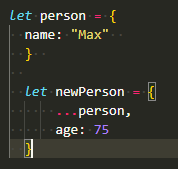
**Spread and rest operators**

Оператор …**spread**:

Копируем данные из **массива**:



Копируем данные из **объекта**:



**Destructuring**

Оператор …**spread**:



2. React Basics &

Working with components

Самая важная концепция в реакте это **компоненты**. В конечном итоге все состоит из них. Любой компонент в Реакте это простая ФУНКЦИЯ!

*Компонент:*

*- может быть переиспользован много раз (DRY)*

*- должен отвечать за что-то одно. Не надо реализовывать много функционала в одном компоненте. (SOLID)*

*Как построен компонент?*

Компонент состоит из: HTML, JS, CSS

Компонент должен быть написан в **декларативном стиле.** Тогда как приложение на обычном JS будет написано по большей части в императивном стиле.

1. The Concept of “Composition”

{props.children}

**Композиция – построение приложения по схеме на картинке. Т.е. каждый дочерний компонент явно указывается в родительском.**

Что, если мы хотим создать компонент – обертку? Для этого применяется **декомпозиция.** Например:

import './Card.css'

//now card is a CONTAINER component

*function* Card (*props*) {

  // так можно применять несколько стилей: стиль обертки и стиль родителя.

  // в строке ПРОБЕЛ!!!

*const* classes = 'card ' + *props*.className

  return <div className={classes}>{*props*.children}</div>

}

export default Card

**И обернем в него 2 компонента:**

import ExpenseItem from './ExpenseItem'

import Card from './Card'

import './Expenses.css'

*function* Expenses ({ *items* }) {

  return (

    <*Card* className={'expenses'}>

      <*ExpenseItem*

        title={*items*[0].title}

        amount={*items*[0].amount}

        date={*items*[0].date}

      />

      <*ExpenseItem*

        title={*items*[1].title}

        amount={*items*[1].amount}

        date={*items*[1].date}

      />

      <*ExpenseItem*

        title={*items*[2].title}

        amount={*items*[2].amount}

        date={*items*[2].date}

      />

      <*ExpenseItem*

        title={*items*[3].title}

        amount={*items*[3].amount}

        date={*items*[3].date}

      />

    </*Card*>

  )

}

**И**

import ExpenseDate from './ExpenseDate'

import './ExpenseItem.css'

import Card from './Card'

*function* ExpenseItem (*props*) {

  return (

    <*Card* className='expense-item'>

      <*ExpenseDate* date={*props*.date} />

      <div className='expense-item\_\_description'>

        <h2>{*props*.title}</h2>

      </div>

      <div className='expense-item\_\_price'>${*props*.amount}</div>

    </*Card*>

  )

}

export default ExpenseItem

**Зачем:**

* **Reusable component с набором стилей и без дочерних компонентов**

1. React State and Events

Если нужно обновлять **стейт с использованием предыдущего состояния** (например, **если** **стейт это объект**), то в функцию, сетающую стейт передать не просто значение, как обычно, а колл-бек:

const titleChangeHandler = (*event*) => {  
 setUserInput((*prevState*)=>({...*prevState*, enteredTitle: *event*.target.value}))  
};

Обычно мы делаем так:

const amountChangeHandler = (*event*) => {  
 setUserInput(*event*.target.value)  
};

Two-way binding

Чтобы не возникало ошибки:

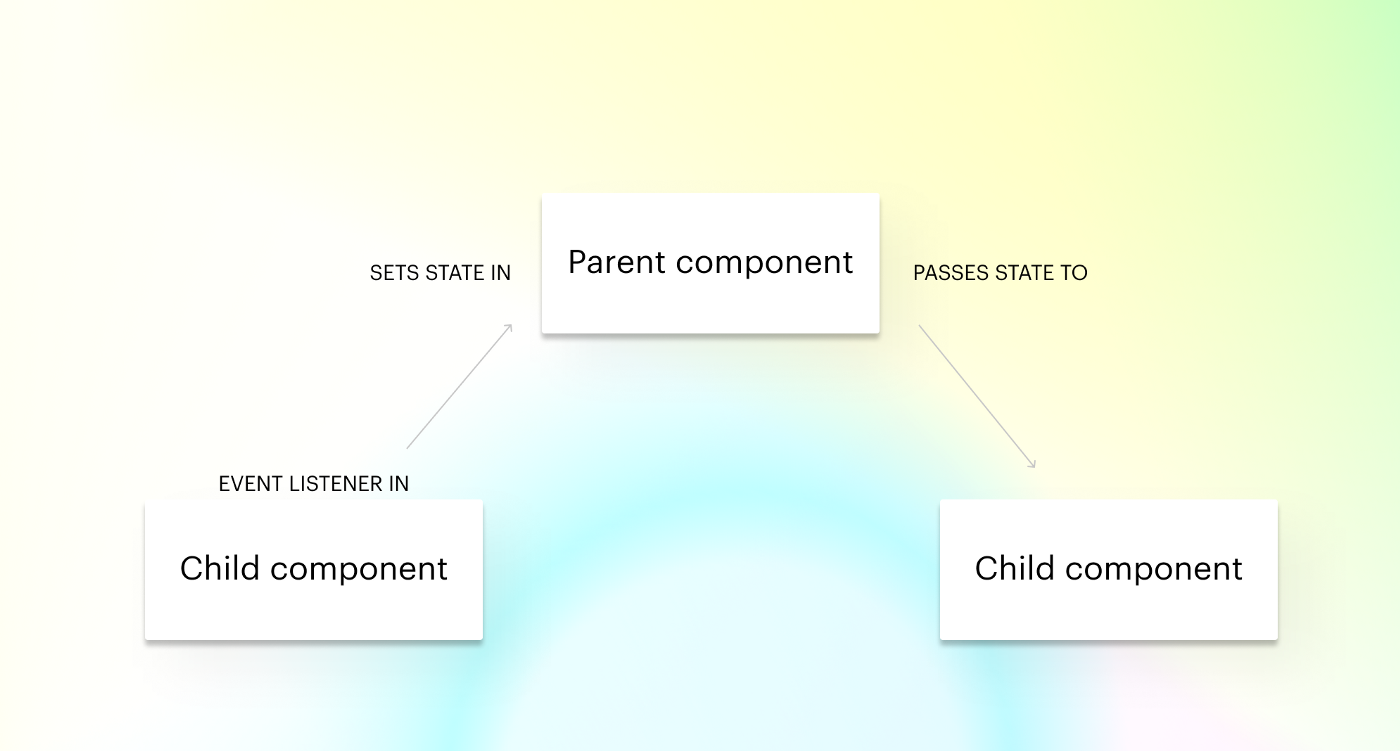
**Warning: A component is changing a controlled input to be uncontrolled. This is likely caused by the value changing from a defined to undefined, which should not happen. Decide between using a controlled or uncontrolled input element for the lifetime of the component.**

Она возникает из-за того, что стейт в первом рендере может быть Undefined.

Нужно прописать в value значение, которое будет отличным от Undefined, а именно:

return (  
 <form onSubmit={submitHandler}>  
 <div className={"new-expense\_\_controls"}>  
 <div className={"new-expense\_\_control"}>  
 <label>**Title**</label>  
 <input type='text' value={userInput.enteredTitle || ''} onChange={titleChangeHandler}/>

Подъем состояния



Данные собираются в дочернем компоненте и через колл-бек возвращаются к родителю. Там они сетаются в стейт и передаются через пропсы во второй дочерний компонент.

Controlled vs Uncontrolled Components

Stateful vs Stateless Components

**Controlled component** – принимает *от родителя* колл-бек через пропсы, передает в него данные, обработанные данные *получает опять от родителя через пропсы*.

**Stateless components** (presentational, dumb) – компонент, у которого нет своего стейта.

1. Rendering Lists & Conditional content

**Rendering Lists** - на входе массив, затем маппинг массива

**Conditional Content** – можем использовать:

1. Тернарные операторы (if else нельзя!)  **«…?...:»**
2. Логические операторы && , ||
3. Вынести JSX разметку в отдельную переменную:

let expensesContent = <p>**No expenses found**</p>;

и использовать ее далее:

function Expenses({*items*}) {  
  
 const [filteredYear, setFilteredYear] = useState('2020');  
  
 const filterChangeHandler = *selectedYear* => {  
 setFilteredYear(*selectedYear*)  
 };  
  
 const filteredExpenses = *items*.filter(*item* => *item*.date.getFullYear() === +filteredYear);  
  
 **let expensesContent = <p>No expenses found</p>;  
  
 if (filteredExpenses.length > 0) {  
 expensesContent = filteredExpenses.map(*expense* => (  
 <ExpenseItem  
 key={*expense*.id}  
 title={*expense*.title}  
 amount={*expense*.amount}  
 date={*expense*.date}/>)  
 )  
 }**  
 return (  
 <div>  
 <Card className={'expenses'}>  
 <ExpensesFilter

selected={filteredYear} onChangeFilter={filterChangeHandler}/>  
 {expensesContent}  
 </Card>  
 </div>  
 )  
}  
  
export default Expenses

1. Сделать **два и более** **отдельных return** в функциональном компоненте. Срабатывает первый, в котором выполняется условие.

if (!showExpenseForm) {

  return (

    <div className={'new-expense\_\_controls'}>

        <button onClick={() *=>* showExpenseFormHandler(true)}>

          Add New Expense

        </button>

    </div>

  )

}

  return (

        <form onSubmit={submitHandler}>

          <div className={'new-expense\_\_controls'}>

            …

          </div>

          <div className={'new-expense\_\_actions'}>

            <button onClick={() *=>* showExpenseFormHandler(false)}>Cancel</button>

            <button type='submit'>Add expense</button>

          </div>

        </form>

  )

}

export default ExpenseForm

Adding Dynamic Styles

Если нужно изменять значение стилей динамически, изменяемую переменную можно передать в **атрибут style**:

*const* ChartBar = *props* *=>* {

*let* barFillHeight = '0%'

  if (*props*.max > 0) {

    barFillHeight = Math.round((*props*.value / *props*.maxValue) \* 100) + '%'

  }

  return (

    <div className={'chart-bar'}>

      <div className={'chart-bar\_\_inner'}>

        <div className={'chart-bar\_\_fill'} style={{height: barFillHeight}}></div>

      </div>

      <div className={'chart-bar\_\_label'}>{*props*.label}</div>

    </div>

  )

}

export default ChartBar

1. Styling React Components

Если нужно менять стиль элемента динамически, т.е. после действия пользователя на странице, например, делаем следующее:

1. Создаем флаг в стейте(или просто стейт)
2. (ПЛОХАЯ ПРАКТИКА)Далее можно через инлайн стили задать:

<label style={{ color: !isValid ? 'red' : 'balck' }}>Course Goal</label>

1. (ХОРОШАЯ ПРАКТИКА):

- Создаем нужный класс в css:

.form-control.invalid input {

*border-color*: red;

*background-color*: #ffd7d7;

}

.form-control.invalid label {

*color*: red;

}

* Далее используем шаблонные строки:

  <div className={`form-control ${!isValid? 'invalid' : ''}`}>

Так можно добавлять через $ и другие переменные со стилями!

Using CSS Modules

Чтобы использовать СSS Modules в проекте, нужна его поддержка. При создании приложения с помощью create react app такая поддержка устанавливается автоматически

Обозначение:

1. Просто стиль при импорте обычного сss:

**className=”wrapper”**

1. При использовании модулей:

**className={styles.wrapper}**

**className={styles[‘form-control’]}** – если есть запрещенные символы

1. Динамическая передача стилей (template strings)

 <div className={`${styles['form-control']} ${!isValid && styles.invalid}`}>

1. Debugging React Apps

React Dev Tools (Chrome extension)

В расширении во вкладке Components можно посмотреть:

- Пропсы, которые приходят в компонент

- Коллбеки

- Хуки, которые используются (**можно увидеть текущий стейт**). Стейт можно править прямо в расширении

- Вложенность компонент

1. Building a project
2. Fragments, Portals, Refs

React.Fragment

Любой код в JSX должен иметь **одного общего родителя:**

 return (

    <div>

      {error && (

        <*ErrorModal*

          title={error.title}

………………………………………}

            onChange={ageChangeHandler}

          />

          <*Button* type="submit">Add User</*Button*>

        </form>

      </*Card*>

    </div>

  );

};

1. **Вариант: обернуть все в <div>**

return (

    <div>

…

    </div>

  );

1. **Вариант: обернуть все в массив [ ]. Но! Надо передавать атрибут key! Также проставить запятые после элементов!**

return (

    [

…

    </div>

  ];

1. **Вариант: обернуть все в React Fragment < >. Это по сути пустой компонент. Good practice!**

return (

    <React.Fragment>

…

    </div>

  < React.Fragment />;

{Fragment} в этом случае надо импортировать из React

Новый синтаксис:

return (

    <>

…

    </div>

  </>;

React Portals

из react-dom

Проблема: код всплывающего модального окна находится где-то внутри разметки документа, т.е. все лежит вложено в root дивке. И это будет работать. Но семантически правильнее не мешать модальное окно и остальной контент. Т.е. одно не должно быть вложено в другое, а быть siblings. Для решения этой проблемы используются react.fragment.

Для использования Portals нужно две вещи:

1. Место, куда нужно портировать компонент.
2. Дать компоненту знать, что у него есть портал в это место

Для этого:

1. Идем в public/index.html
2. Добавляем над id=”root” дивы, соответствующие тем, куда мы хотим портировать компонент.

<body>

    <noscript>You need to enable JavaScript to run this app.</noscript>

    <div id="backdrop-root"></div>

    <div id="modal-root"></div>

    <div id="root"></div>

    <!--

      This HTML file is a template.

      If you open it directly in the browser, you will see an empty page.

      You can add webfonts, meta tags, or analytics to this file.

      The build step will place the bundled scripts into the <body> tag.

      To begin the development, run `npm start` or `yarn start`.

      To create a production bundle, use `npm run build` or `yarn build`.

    -->

  </body>

1. Импортируем reactDOM

import ReactDOM from 'react-dom'

1. Создаем портал. Не забудь передать пропсы!

*const* ErrorModal = *props* *=>* {

  return (

    <>

      {ReactDOM.createPortal(

        <*Backdrop* onConfirm={*props*.onConfirm} />,

        document.getElementById('backdrop-root')

      )}

      {ReactDOM.createPortal(

        <*ModalOverlay*

          title={*props*.title}

          message={*props*.message}

          onConfirm={*props*.onConfirm}

        />,

        document.getElementById('modal-root')

      )}

    </>

  )

}

Итого так выглядит компонент:

import React from 'react';

import Card from './Card';

import Button from './Button';

import classes from './ErrorModal.module.css';

import ReactDOM from 'react-dom'

*const* Backdrop = *props* *=>* {

  return  <div className={classes.backdrop} onClick={*props*.onConfirm} />

}

*const* ModalOverlay = *props* *=>* {

  return (

    <*Card* className={classes.modal}>

    <header className={classes.header}>

      <h2>{*props*.title}</h2>

    </header>

    <div className={classes.content}>

      <p>{*props*.message}</p>

    </div>

    <footer className={classes.actions}>

      <*Button* onClick={*props*.onConfirm}>Okay</*Button*>

    </footer>

  </*Card*>

  )

}

*const* ErrorModal = *props* *=>* {

  return (

    <>

      {ReactDOM.createPortal(

        <*Backdrop* onConfirm={*props*.onConfirm} />,

        document.getElementById('backdrop-root')

      )}

      {ReactDOM.createPortal(

        <*ModalOverlay*

          title={*props*.title}

          message={*props*.message}

          onConfirm={*props*.onConfirm}

        />,

        document.getElementById('modal-root')

      )}

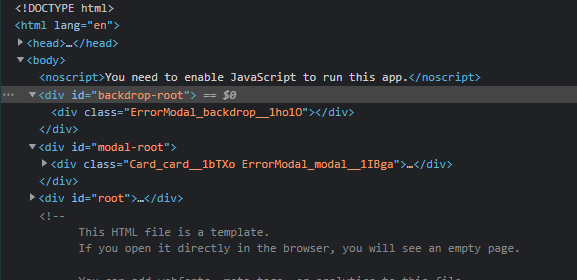
    </>

  )

}

export default ErrorModal;

Теперь наши два компонента находятся вне root дива и ВСЕГДА будут там находиться.



Working with REFs

Например, у нас есть инпут, данные из которого мы будем брать только при сабмите, тогда зачем нам каждый раз ренедерить этот инпут, если мы делаем two-way binding? Можно просто взять данные из рефа.

1. **Создаем реф**

*const* ageInputRef =useRef()

1. **Передаем в реф атрибут в инпут** и удаляем value и onChange (если были)

          <input

            id="username"

            type="text"

            ref={nameInputRef}

          />

1. В функции, которая обрабатывает **данные из инпутов, выбираем** **данные из рефа**

*const* enteredName = nameInputRef.current.value

*const* enteredUserAge = ageInputRef.current.value

1. **Сбрасывать значение** инпутов можно напрямую манипулируя ДОМ (но в других случаях напрямую манипулировать ДОМ плохо)

  nameInputRef.current.value = ''

    ageInputRef.current.value = ''

В этом случае мы сделали компонент НЕКОНТРОЛИРУЕМЫМ, т.к. мы не контролируем компонент с помощью реакта.

Если бы мы контролировали инпут с помощью useState, тогда это был бы КОНТРОЛИРУЕМЫЙ элемент

**Когда еще можно использовать реф?**

**- Если нужно просто быстро прочитать значение без сохранения состояния с рендером**

1. Advanced: Handling Side Effects, Using Reducers & Using the Context API

useEffect()

В качестве зависимости в **useEffect** можно также передавать функции

**НЕ ДОБАВЛЯТЬ в зависимости useEffect:**

- state updating functions из useState

- встроенные API (fetch(), localStorage и пр.)

- функции и переменные, определяемые ВНЕ компонента(функции)

Clean up в useEffect()

**useEffect срабатывает при каждом цикле рендера компонента, а также при его монтировании**

**- нет массива зависимостей:** срабатывает при каждом рендере

**- пустой массив зависимостей []** : срабатывает только при монтировании

**- с зависимостями в массиве [dep1, dep2]:** при монтировании и при изменении зависимости.

**Clean up function** вызовется каждый раз **ДО** того, как useEffect вызовет основную функцию!!! (кроме первого раза, конечно) А также, ПЕРЕД тем, как компонент будет демонтирован из DOM

  useEffect(()*=>*{

*const* identifier = setTimeout(()*=>*{

      console.log('Cheking from validity');

      setFormIsValid(

        enteredEmail.includes('@') && enteredPassword.trim().length > 6

        );

    }, 2000);

    return () *=>* {

      console.log('CLEANUP')

      clearTimeout(identifier)

    }

  }, [enteredEmail, enteredPassword])

useReducer()

Хук useReducer обычно предпочтительнее useState, **когда у вас сложная логика состояния**, которая включает в себя несколько значений, или когда следующее состояние зависит от предыдущего. useReducer также позволяет оптимизировать производительность компонентов, которые запускают глубокие обновления, [поскольку вы можете передавать dispatch вместо колбэков](https://ru.reactjs.org/docs/hooks-faq.html#how-to-avoid-passing-callbacks-down).

const [state, dispatch] = useReducer(reducer, initialState, initialFunction);

**state** – текущее состояние

**dispatch** – функция, с помощью которой диспатчится новое состояние, т.е. триггер для обновления состояния.

**Reducer –** функция, которая вызывается автоматически, когда задиспатчен экшен, и возвращает новое состояние (функция-преобразователь состояния)

**initialState** - начальное состояние (initialState)

**initialFunction** – функция, которая сетает начальное состояние **(например, когда начальный стейт является результатом обработки http запроса)**

useContext()

Используется для того, чтобы **достать данные из Context**. *Ранее* для этого использовались Context.consumer и Context.Provider, MSTP и MDTP.

***Когда использовать useContext?***

Полезно создавать отдельные кастомные контексты, такие как AuthContext, locale, theme . Завернуть в него все приложение и доставать/передавать в него нужные данные

**Стоит запомнить:**

* Контекст **не** является полной заменой пропсов и использования хука useState(), и Redux, т.к. при излишнем использовании контекста компонент может стать не пригодным для многократного использования
* Контекст не оптимизирован для очень частых изменений состояния (чаще одного раза в минуту-секунду)

**Правила всех хуков:**

- **Хуки можно** вызывать только в React Function Components и в Custom Hooks

- **Хуки можно** вызывать только на верхнем уровне функциональной компоненты.

*- UNOFFICIAL: надо добавлять все переменные, используемые в* ***useEffect****, в массив зависимостей?? Кроме значений/функций-сеттеров стейта (из хука useState)(…почему??)*

**ХУКИ НЕЛЬЗЯ**:

- Вызывать внутри другого хука (например в useEffect)

- Внутри другой вложенной функции (например, onClickHandler)

- Внутри block statement (например if…else)

Forward Refs

(хук useRef)

*Используется, если нужно напрямую воздействовать на DOM элемент. Например, для нативных элементов (input, button и пр.) вызывать встроенные методы, такие как focus() и пр.*

1. Создать реф:

*const* inputRef = useRef()

1. Передаем Ref как пропс в input элемент:

!Нельзя передавать таким образом ref в React компоненты, только в нативные DOM (input, button). Чтобы прокинуть ref в функциональный компонент, надо использовать хук **useImperativeHandle()**

return <input ref={inputRef} />

1. Добавляем в useEffect например фокус на инпут при монтировании компонента (либо можем использовать его в функции для активации кнопки). Тем самым мы напрямую воздействуем на DOM. Это не очень хорошая практика, но иногда других вариантов нет. Метод focus() – встроенный в JS

  useEffect =

    (() *=>* {

      inputRef.current.focus()

    },

    [])

1. Practice Project
2. A look behind the scenes & Optimization

**Рендер компонента(re-evaluation) происходит только когда:**

1. Меняются пропсы

2. Меняется контекст, если компонент подписан на его обновление

3. Меняется стейт useState()

4. Меняется стор, если компонент подписан на обновление этой части стора

Если меняется стейт РОДИТЕЛЯ, то происходит re-evaluation(рендер) всех дочерних компонент!!! **Даже если пропсы у дочернего компонента не поменялись. Для того, чтобы это предотвратить, нужно использовать МЕМОИЗАЦИЮ**

**ОПТИМИЗАЦИЯ**

**!!!Все способы оптимизации также требуют затрат ресурсов, нужно точно понимать, что затраты на мемоизацию меньше, чем на лишний рендеринг компонент!**

**!!Стоит понимать,** что рендер компонента — это просто перезапуск функции, и при каждом запуске функции все переменные, объекты, функции создаются заново. *Все способы мемоизации сравнивают значения, а сравнение примитивов, объектов, функций происходит по-разному. Поэтому мемоизация делается несколькими способами в зависимости от того, какие типы данных будем сравнивать при повторном рендере:*

1. Если props являются **примитивами**, то для предотвращения рендеринга, вызванного рендером родителя можно использовать:

**React.memo() HOC**– оборачиваем дочерний компонент в объявлении или в export default

const ToTheMoonComponent = React.memo(function MyComponent(props) {

    // повторно отображается только если изменилось свойство

});

1. Если props является **функцией**, то заворачиваем объявление функции в родителе в хук:

**useCallback() hook**

const memoizedCallback = useCallback( () => { doSomething(a, b); }, [a, b],);

**В массив зависимостей useCallBack** добавляем все переменные, которые используем в функции, т.е. при изменении которых будем говорить Реакту, что функция изменилась.

В useCallback можно добавлять например, handlers

1. Создать **любое мемоизированное значение**, которое не будет пересчитываться при каждом рендере:

**useMemo () hook**

const memoizedValue = useMemo(() => computeExpensiveValue(a, b), [a, b]);

1. Class Based Components

**Простой синтаксис:**

class Welcome extends React.Component {

render() {

return <h1>Привет, {this.props.name}</h1>;

}

}

Чтобы использовать **стейт**, надо использовать Constructor:

class Calculator extends React.Component {

constructor(props) {

super(props);

this.handleChange = this.handleChange.bind(this);

this.state = {temperature: ''}; } // СТЕЙТ, в классовой компоненте это ВСЕГДА ОБЪЕКТ. В функциональной – любой тип данных

handleChange(e) {

this.setState({temperature: e.target.value}); } //изменять стейт можно только в методе SetState

render() {

const temperature = this.state.temperature;

return (

<fieldset>

<legend>Введите температуру в градусах Цельсия:</legend>

<input value={temperature} onChange={this.handleChange} /> <BoilingVerdict celsius={parseFloat(temperature)} /> </fieldset>

);

}

}

ВАЖНО в классовых компонентах:

1. Чтобы использовать **стейт**, надо использовать Constructor:
2. СТЕЙТ, в классовой компоненте это ВСЕГДА ОБЪЕКТ. В функциональной – любой тип данных
3. Изменять стейт можно только в методе SetState (принимает только объект!). При этом происходит **не** полная замена объекта (merge), а только совпадающий полей!!! В хуке – полная замена
4. Методы надо всегда байндить в методе render()!

*this.handleChange = this.handleChange.bind(this);*

1. Super(props); - наследование конструктора из класса Component \_ обязательно

**Методы жизненного цикла**

**ComponentDidMount()** – useEffect(…, [пустой массив])

**ComponentDidUpdate(prevProps, prevState)** – useEffect(…, [зависимости]). *Чтобы не было бесконечного цикла апдейта стейта, надо прописывать if в этом методе, который сравнивает prevProps, prevState с текущими state и props. Это обусловлено тем, что этот метод срабатывает при каждом изменении стейта или пропсов, поэтому и изменение стейта надо делать по условию if.*

**ComponentWillUnmount() –** useEffect(()=> {return=>{…}, []}) (**cleanUp func**) – выполняется ПЕРЕД тем, как будет запущен колл бек в useEffect и перед тем, как будет элемент удален из DOM! *Срабатывает, например когда мы отображаем компонент с помощью conditional rendering.*

**Context в классовых компонентах**

В функциональных компонентах мы используем хук useContext и можем быть подписаны на множество контекстов. В классовых компонентах мы можем использовать **только один context. Подключается контекст путем создания публичного статического поля static в классовой компоненте:**

**static contextType = usersContext**

(Публичные статические поля не переопределяются в наследниках класса, а могут быть доступны через иерархию прототипов.)

Создание контекста происходит как в функциональной компоненте:

import React from 'react';

const UsersContext = React.createContext({

users: []

});

export default UsersContext;

При этом доступ к контексту осуществляется с помощью **this.context**

**Error Boundaries**

**(нет эквивалента в функциональных компонентах)**

Служит для обработки ошибок.

Например мы бросаем ошибку при каких-то условиях: throw new Error(‘’No users”). Если не обработать эту ошибку, то она крашнет все приложение. Обычно для этого в JS используется конструкция try… catch, но если ошибка вылетает при неверных пропсах от родителя, то завернуть JSX в эту конструкцию не получится.

1. Создаем компонент, например ErrorBoundary.js. В нем определяем, как будем обрабатывать ошибку.

class ErrorBoundary extends React.Component {

constructor(props) {

super(props);

this.state = { hasError: false };

}

static **getDerivedStateFromError**(error) {

// Update state so the next render will show the fallback UI.

return { hasError: true };

}

**componentDidCatch**(error, errorInfo) { **// ОБРАБОТКА ОШИБКИ**

// You can also log the error to an error reporting service

logErrorToMyService(error, errorInfo);

}

render() {

**if (this.state.hasError) {**

// You can render any custom fallback UI

return <h1>Something went wrong.</h1>;

}

return **this.props.children;**

}

}

1. Обернуть компонент, в котором будет ловиться ошибка в ErrorBoundary (используется this.props.children)
2. HTTP Requests
3. Custom React hooks

**Custom hook** – Это по сути функция, которую мы будем использовать повторно и у которой есть свое состояние.

**Когда использовать?**

- Используются для повторяемой логики (re-using logic), **когда в этом коде есть реактовские хуки. Другими словами – если есть кусок кода, который я хочу переиспользовать, и в нем есть реактовские хуки – надо создавать не просто функцию (хуки в ней использовать нельзя), а создавать custom hook.**

- В них можно использовать обычные хуки

- Называть надо с префиксом “use”

Пример:

import { useState, useEffect } from 'react';

function useFriendStatus(friendID) {

const [isOnline, setIsOnline] = useState(null);

useEffect(() => {

function handleStatusChange(status) {

setIsOnline(status.isOnline);

}

ChatAPI.subscribeToFriendStatus(friendID, handleStatusChange);

return ( ) => {

ChatAPI.unsubscribeFromFriendStatus(friendID, handleStatusChange);

};

}, []);

return isOnline; **// возвращаем значение стейта, которое будем использовать в компоненте для рендера**

}

…

26. Testing React Apps

Тестирование:

**Ручное тестирование** (пишу код и смотрю, что получилось)

**Автоматизированное тестирование** (запускаю код тестов, они проверяют все модули)

**Типы автоматизированного тестирования:**

**Unit tests –** написание тестов для самых маленьких отдельных юнитов («строительных блоков»: функции, компоненты, классы) в изоляции друг от друга. Обычно проект содержит сотни юнит-тестов.

**Integration tests –** проверяется взаимодействие отдельных юнитов между собой. Тестируется, например, как компоненты взаимодействуют между собой. Их обычно меньше, чем юнит-тестов. Иногда нет явной разницы в определении типа тестирования – юнит или интеграционных в реакте.

**End-to-End (e2e) tests –** тестируются сценарии взаимодействия юзера с приложением, т.е. тестируется как бы поведение пользователя. Их обычно меньше, чем интеграционных тестов.

**Tools & Setup**

Нам нужны:

- Инструмент для запуска тестов и назначения результатов (**Jest**)

- Симуляция (рендеринг) тестируемых компонент (**React Testing Library**)

*Оба этих инструмента уже есть в зависимостях при создании приложения с помощью CRA!*

**Общий синтаксис**

**test(‘description’, ()=>{**

**})**

**ААА:**

A – **arrange** (задаем тестовые данные, тестовые условия, окружение)

А – **act** (выполнить логику, которая должна быть протестирована. Например, запустить выполнение функции)

А – **assert** **the result** (сравнение полученных результатов с ожидаемыми)

**Пример:**

    test('renders "Changed!" if the button was clicked', () => {

        // Arrange

        render(<Greeting />)

        // Act

        const buttonElement = screen.getByRole('button')

        userEvent.click(buttonElement)

        //Assert

        const outputElement = screen.getByText('Changed!')

        expect(outputElement).toBeInTheDocument()

    })

Несколько тестов можно объединять в test suites. Создается с помощью функции describe. Например:

describe('<Greeting/> component', () => { **// создает отдельный suite**

    test('renders Hello World', () => {

        // Arrange

        render(<Greeting />)

        // Act

        // ...nothing

        //Assert

        const helloWorldElement = screen.getByText('Hello, World')

        expect(helloWorldElement).toBeInTheDocument()

    })

})

**Тестирование user interaction и State**

Для тестирования поведения пользователя будем использовать библиотеку React Testing Library.

В данном примере будем использовать метод **userEvent()**

    test('renders "Changed!" if the button was clicked', () => {

        // Arrange

        render(<Greeting />)

        // Act

        const buttonElement = screen.getByRole('button') **// находим элемент**

        userEvent.click(buttonElement) **// имитируем нажатие на кнопку**

        //Assert

        const outputElement = screen.getByText('Changed!')

        expect(outputElement).toBeInTheDocument()

    })

**Тестирование взаимосвязанных компонент**

Тест, который мы написали проверяется **все дерево компонентов**, начиная с родителя и ниже. По этой причине мы можем сделать почти **все тесты в родительской компоненте**. Если же родительские компоненты достаточно сложны, можно написать для них тесты отдельно. Т.е. если в компонент Greeting добавить дочерние компоненты с пропсами, тест тоже будет проходить.

**Тестирование АСИНХРОННОГО кода**

Например, у нас есть компонент с асинхронным кодом:

const Async = () => {

  const [posts, setPosts] = useState([]);

  useEffect(() => {

    fetch('https://jsonplaceholder.typicode.com/posts')

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

      .then((data) => {

        setPosts(data);

      });

  }, []);

  return (

    <div>

      <ul>

        {posts.map((post) => (

          <li key={post.id}>{post.title}</li>

        ))}

      </ul>

    </div>

  );

};

export default Async;

Два варианта тестирования асинхронного кода:

1. **Асинхронные тесты (минусы: во время тестов отправляется много лишних запросов на сервер, во время POST запросов будут модифицироваться данные)**
2. **Использование МОКОВ (mock function)**

**Асинхронные тесты**

**(с async/await и ‘find…’)**

В этом случае вместо методов **‘get’** надо использовать методы ‘**find’**, т.к. в первом случае машина ищет элементы сразу при первом рендеринге, синхронно. Но в нашем случае апдейт стейта происходит асинхронно, после первого рендера компонента. Именно поэтому обычный метод с ‘get’ покажет ошибку.

Методы с ‘find’ возвращают ПРОМИС, поэтому колл-бэк метода test() надо заворачивать в async/await

В методе можно задавать exact: true/false и таймер (см. синтаксис)

Напишем тест для асинхронного кода с find:

  test('renders posts if requests succeeds', async () => {

        render(<Async />)

        const listItemElements = await screen.findAllByRole('listitem')

        expect(listItemElements).not.toHaveLength(0)

    })

**Использование МОКОВ**

**(mock function)**

Используем вместо данных замоканый объект:

 test('renders posts if requests succeeds', async () => {

        window.fetch = jest.fn()

        window.fetch.mockResolvedValueOnce({

            json: async () => ([

                { id: 'p1', title: 'First post' } // замоканый объект

            ])

        })

        render(<Async />)

        const listItemElements = await screen.findAllByRole('listitem')

        expect(listItemElements).not.toHaveLength(0)

    })

**Итого:** для тестирования Реакта нам нужно использовать две библиотеки: Jest и React Testing Library. Обе уже установлены в CRA!

Также может быть полезной библиотека **react-hooks-testing-library** для тестирования хуков и кастомных хуков реакта.

27. React + TYPESCRIPT

TS добавляется СТАТИЧЕСКОЕ типизирование в JS. В JS изначально динамически типизированный язык.

Как использовать TS в Nodejs? (ПРИМЕР «РУЧНОЙ» КОМПИЛЯЦИИ)

- Устанавливаем сам typescript:

*npm install typescript / install -g typescript*

- Запускаем компилятор tsc:

*npx tsc (если есть файл конфига, в котором прописаны папки, которые нужно скомпилить, то норм. Если нет конфига, то пишем просто файл, который нужно перекомпилить, как ниже)*

- Компилируем файл js из ts. После этого компилятор tsс на основе существующего файла ts создаст файл js:

*npx tsc with-typescript.ts*

**UNION Types**

Union types – это тип данных, который может быть ОДИН ИЗ НЕСКОЛЬКИХ типов, определяется помощью оператора «|». Например:

let course: string | number = ‘Hello’

**Unions with Common Fields**

Если объединить два интерфейса с помощью Union Type, то доступ есть только к элементам, общим для обоих типов () !!!

Поэтому, чтобы использовать union как «или», надо сделать одно для всех общее поле с конкретным строковым значением. Так Typescript определит, какой тип ты используешь (см. https://www.typescriptlang.org/docs/handbook/unions-and-intersections.html)

interface Bird {

fly(): void;

layEggs(): void;

}

interface Fish {

swim(): void;

layEggs(): void;

}

declare function getSmallPet(): Fish | Bird;

let pet = getSmallPet();

pet.layEggs();

// Only available in one of the two possible types

pet.swim();

Property 'swim' does not exist on type 'Bird | Fish'. Property 'swim' does not exist on type 'Bird'.

**INTERSECTION Types**

Intersection types – это тип данных, который объединяет несколько типов данных в себе с помощью оператора «&».

См. <https://stackoverflow.com/questions/35726858/typescript-union-types-dealing-with-interfaces>

OPERATORS

**Question mark ?** – означает опциональный параметр, т.е. в тип данных добавляется Undefined. Например, string | undefined. В цепочке доступа к свойствам объекта (*optional chaining (?.) operator)* дает возможность избежать ошибки “reading of undefined”

**Exclamation mark ! (non-null assertion)**  - убирает типы null и undefined из типов данных переменной. Можно избежать ошибки object if possibly null/undefined, но в этом случае надо быть на 100% уверенным, что оно так и есть (см. примеры ниже).

**проверка на Null / undefined**

type Employee = {

id: number;

name: string;

};

function getEmployeeName(emp?: Employee) {

***// ⛔️ Error: Object is possibly 'undefined'.ts(2532)***

return emp.name;

}

*// 👇️ "Frank"*

console.log(getEmployeeName({ id: 1, name: 'Frank' }));

**Варианты решения проблемы (проверка на Null / undefined):**

**1) Использование «!»**  
  
function getEmployeeName(emp?: Employee) {

return emp**!**.name; **// 👈️ use non-null assertion (мы явно говорим – значение точно не null или undefined)**

}

1. **Использование «as»**

function getEmployeeName(emp?: Employee) {

return (emp **as** Employee).name; **// 👈️ type assertion (мы говорим – эта переменная точно будет соответствовать типу Eployee)**

}

1. **Использование «?»**

function getEmployeeName(emp?: Employee) {

return emp**?**.name; **// 👈️ use optional chaining (здесь мы говорим – да, параметр опциональный)**

}

1. **Проверка через «if»**

function getEmployeeName(emp?: Employee) {

**if** (emp) {

**// 👉️ emp is type Employee here (здесь мы говорим – проверь, если есть это значение, то выведи его )**

return emp.name;

}

**// 👉️ emp is type undefined here**

return 'James Doe';

}