* **Virtual DOM PROCESS** – **Virtual DOM** обновляется при изменениях. Вычисляется разница между новым Virtual и Real DOM. Обновляет только измененную часть Real DOM.
* **render()** – возвращает **React** элемент, который является компонентом DOM.
* **events** – используется обертка **SyntheticEvent** – все обработчики событий получают ее экземпляр. У нее такой же интерфейс включая stopPropagation() и preventDefault(). Помогает событиям работать во всех браузерах одинаково. Можно обратиться к нативному эвенту через **event.nativeEvent**. Обработчики вызываются на стадии всплытия(bubbling). Добавить onClick**Capture**, чтобы обработать событие на фазе перехвата. Не нужно вызывать addEventListener. **Буфер**. onCopy onPaste. **Клавиатура** onKeyDown onKeyPress. **Фокус** onFocus onBlur. **Форма** - onChange onInput onInvalid. **Общие** onError onLoad. **Мышь** - onDoubleClick onDrag onDragEnd onDrop onMouseEnter onMouseMove onMouseOver. **Курсор** onPointerDown onPointerMove onGotPointerCapture. **Выбор** onSelect. **UI** onScroll. **Анимация** onAnimationStart onAnimationEnd onAnimationIteration и **другие события** в документации.
* **refs** – ссылки, помогает хранить ссылку на конкретный элемент DOM или компонент React. Можно получить данные об объекте по ссылке. Пригодится для управления фокусом, выделения текста, воспроизведения мультимедиа, запуска анимации, интеграция со сторонними библиотеками.
* **keys** – необходимы для все списков в React. Используются для идентификации, что повышает производительность. Можно присваивать индекс при генерации, но лучше хранить на стороне данных, чтобы id действительно были уникальны.
* **<ReactFragment><**/ **ReactFragment**> **<><**/> - позволяет добавлять прозрачный контейнер. Избегая дополнительных div

**State Management Tools**

* **Different size of state:**
  + **small state – useState, Context**
  + **medium state – Recoil**
  + **Large state – Redux, MobX**
* **Props drilling** – передача props через большое число компонент, которым они не нужны к конечной компоненте получателю.
  + Пути решения проблемы:
    - Необходимо держать **state** так близко к компоненте ка это **возможно**. Следует избегать излишней разбивки компонент. Но это не значит держать все внутри одной компоненты. Вместо этого следует **передать** **саму** **компоненту** **вместе** **с** необходимыми ей **props**.
    - Использовать React context API. В малых и средних приложениях предпочтительно использовать его, чтобы не перегружать приложение Redux или Mobx.
    - Использовать **State management** **библиотеки** – **Redux**, **Mobx**

**Context**

* **Context -** контекст разработан для передачи данных, которые можно назвать «глобальными» для всего дерева **React** компонентов. Позволяет избежать передачи пропсов в промежуточные компоненты. В основе стоят Providers – компоненты хранящие данные(state) и Consumers – компоненты принимающие данные(state). Использование:
  + 1. **Создать** отдельные файл **SomeContext.js**, в котором создать глобальный контекст:

import { createContext } from 'react';

export const CounterContext = createContext();

* + 2. **Создать** в отдельном файле **Context.Provider**. Зачастую в нем понадобится useState():

import React, { useState } from 'react';

import { CounterContext } from './CounterContext';

export const CounterProvider = ({ children }) => {

  const [numberOfClicks, setNumberOfClicks] = useState(0);

  const increment = (amount) => {

    setNumberOfClicks(numberOfClicks + 1);

  };

  return (

    <CounterContext.Provider value={{ numberOfClicks, increment }}>

      {children}

    </CounterContext.Provider>

  );

};

* + 2\*. Также можно просто обернуть в <**Context.Provider** **value**={{ someValue, orSomeObj }}> без использования отдельного файла, в случае простого контекста.
  + 3. **Передача доступа.** Оборачиваем нужную компоненту в **Context.Provider** (CounterProvider):

import { CounterButton } from './CounterButton';

import './App.css';

import { CounterProvider } from './CounterProvider';

const App = () => {

  return (

    <CounterProvider>

      <h1>State Management Example</h1>

      <CounterButton />

    </CounterProvider>

  );

};

export default App;

* + 4. **Использование. Теперь достаточно вызвать useContext**(SomeContext) хук и передать в него контекст:

import { useContext } from 'react';

import { CounterContext } from './CounterContext';

export const CounterButton = () => {

  const { numberOfClicks, increment } = useContext(CounterContext);

  return (

    <>

      <p>You have clicked the button {numberOfClicks} times.</p>

      <button onClick={() => increment()}>Click</button>

    </>

  );

};

**Flux**

* **Flux** – архитектурный **паттерн, не библиотека как Redux** обеспечивающий однонаправленный поток данных – данные всегда исходят from **view** to the **modal**. Использует
  + **Store** – JS объект для хранения данных во вложенных **stores(в отличии от redux в flux их может быть несколько)**. Предоставляет методы для доступа к состоянию **и** методы к его изменению.
  + Мутация и получения этих данных происходит только посредством **reducers**. **Reducer** - чистая функция, определяет как изменится **state** приложения в ответ на **action** получает данные из **state** или мутируют его опираясь на переданные в него **action-type** и **payload,** которые представляют из себя **action**. Принимает предыдущие состояние **state** и **action.** Возвращает новое состояние **state.**
  + **Action-type** – строка в верхнем регистре. **Payload** – объект с данными. Компонент может передать **action** в **store** посредством функции **action-creator**.

**Redux**

* **Redux –** библиотека для **state** менеджмента, используется для контроля состояния всего **JS** приложения основана на архитектуре **Flux**. Имеет несколько ключевых частей:
  + - **actions –** basically, anything that can happen in our application (when a user clicks on button, when data finishes loading, when data starts loading, any action that could potentially change the state of the entire application is considered an action.) basically action **is an object**:
* export const counterButtonClicked = {
* type: 'COUNTER\_BUTTON\_CLICKED',
* payload: { amount: 1 },
* };
  + - **actions-creators** – basically action is an object. And actions become actions creators when they become functions:
* export const counterButtonClicked = (amount) => ({
* type: 'COUNTER\_BUTTON\_CLICKED',
* payload: amount,
* });
  + - **reducers -**
  + Что отличает **Redux** от других **Flux** библиотек
    - **Redux** **store is** **immutable** – при каждом изменение store мы создаем его новую копию вместо модификации существующего. Это позволяет получать четкий ответ о том, что наше приложение сделало, дает доступ к **undo** and **redo functionality** (функциональность отмены и повтора).
    - Это также означает, что мы **не можем** **использовать** **exampleArray.push(4). Всегда нужно extend the current object,** например используя **spread** оператор: **exampleArray = […exampleArray, 4].**
    - **immutability allow to undo and redo –** это гарантирует отсутствие side эффектов. И это также причина почему мы никогда не редактируем state напрямую:
      * **NOT this: this.**state.booksCompleted = 4 DO this: this.setState({booksCompleted: 4}) – setState вернет новый state.
    - **Store and actions are pure functions** – их легче тестировать и читать. Не будет никаких side effects во время того, как ты dispatch an action. Что ведет к более предсказуемому поведению приложения.
    - **Only one store in Redux** – что значит существует только один источник правды.
  + Состояние хранится в едином **state** – дереве объектов. State предоставляет методы для доступа к состоянию **mapStateToProps**(**state**) => data: selector(state) где selector это указание к полю объекта в state. И методы к его изменению **mapDispatchToProps**(**dispatch**) => **dispatch**(**actionCreator**(**payload**)). **States** управляемых **reducer** может быть несколько, они объединяются при помощи **combineReducers**(). При помощи **configureStore** () можно подключить **persistedReducer**() – который предотвращает **refresh** данных в **store** при обновлении страницы, **composeWithDevTools** – который позволяет использовать **Chrome extensions** и визуально наблюдать работу Redux приложения в браузере, и **applyMiddleware**(**thunk**) который позволяет подключать дополнительные библиотеки для работы с асинхронными запросами, такие как **thunk**, **saga** и др.
  + **Пример.** Если изменить что-то, например в **user profile** в **React** вернется **копия всего старого state** с новыми изменениями не просто модификация. Это может показаться чрезмерным, но есть серьезные преимущества в виде **redo functionality** и **debagging**. Позволяет снизить сложность средних и крупных приложений.
  + **export const configureStore = () => createStore(persistedReducer, composeWithDevTools(applyMiddleware(thunk)));**
  + **Преимущество** **Redux:**
    - **Разделение ответственности**. Предсказуемость результатов (**одно хранилище** – **один источник данных**). Проще в обслуживании при укрупнении приложения – **понятная** **структура**. **Инструменты** **разработчика**. **Тестируемость**.
    - У **Redux** **стандартизирован подход к управлению state**. **State** всегда организован по одному и тому же **pattern**, это означает, что если ты знаешь **Redux** ты легко сможешь перепрыгнуть в абсолютно **новый код** и со старта понимать как происходят **манипуляции** с **данными**.
  + **Недостатки** **Redux:**
    - Каждый раз при вызове **action** создается новый state – это может быть дорогостояще.
* **Immutable.JS** – популярная библиотека которая помогает улучшить производительность. Но у нее, как и у всего есть свои недостатки – нельзя совмещать immutable objects (которые создаются с помощью библиотеки) с обычными JS объектами. Есть пара методов, которые предоставляют взаимодействие, но в обмен на производительность.
* <**Provider** **store**={**store**}> оборачивает <**App** /> компонент для подключения **Redux** **store** к приложению.

**Selectors**

* **Selectors – функции,** хранящиеся в отдельном файле (для удобства), принимают **state как аргумент** и **возвращают** **state.someValue или state**.**someReducer**.**someValue**. Могут быть использованы только в **useSelector(selectorName)** хуке.
  + **Если есть Selectors,** то **useSelector(selectorName) –** импортируемый из react-redux хук для получения необходимых кверей из store.
  + **Если нет Selectors,** то **useSelector**((**state**) => **state**.**someReducer**.**someValue**) – импортируемый из react-redux хук для получения необходимых кверей из store.
* **reselect** – удобная библиотека для создания **selector. import { createSelector } from 'reselect'; альтернатива useSelector() хуку:**
  + **export const getTodos = (state) => state.todos.data;**
  + **export const getIncompleteTodos = createSelector(getTodos, (todos) => todos.filter((todo) => !todo.isCompleted));**
* **useDispatch** – импортируемый из **react-redux** хук **const** **dispatch** = **useDispatch**(); используется для вызова **dispatch** и передачи в него **action-creators**.
* **connect** – функция, из **react-redux**, позволяющая подключить компонент к **Redux**.
  + **export default connect**(**mapStateToProps**, **mapDispatchToProps**)(**Component**);
* **mapStateToProps –** is maps a redux state to props that passed down to the component that it’s connected to.
* **middleware –** структура вызывающая **third-party endpoints,** которые случаются когда мы вызываем **action** и перед тем как **action** достигнет **reducer.** Обычно используются для logging, calling an API, загрузки данных и так далее.
  + **пример middleware. Это прежде всего функция:**
    - **const middleware = store => next => action => { do something } For example:**
* export const logger = (store) => (next) => (action) => {
* console.log('dispatching', action);
* const result = next(action);
* console.log('next state', store.getState());
* return result;
* };
  + - * **store – Redux store**
      * **next function** говорит **redux** продолжитьобработкуследующего **middleware**
      * **action –** последнийаргумент**,** которыйбудетотправленв **reducer**

Во время исполнения **middleware task ты вызовешь next() on the action** что определяет будет ли вызван **middleware** или приложение отправится в **reducer**

**MobX**

* **MobX** - обращается к тем, у кого есть знания в **ООП**. Использует **observer** **patten**, что позволяет системе cтроить **dependency tree** среди разных частей state и позволяет производить свободный рендеринг компонент отслеживая что должно быть мутировано. Дает возможность to manage state, но все равно нужно заниматься организацией кода в форме **stores** и **controllers**. **Store** представляет собой не **JSON объект а spreadsheet with derivations**. Использует концепты **Derivations** and **Reactions**.
  + **Derivation –** любое значение, вычисляемое автоматическииз **state.** Чтобы вызвать **Derivation,** нужноиспользовать **decorator. Decorator** позволяет быть добавленным к объекту без изменения других обьектов.
  + **Плюсы MobX**:
    - нет привязки к организационной структуре использования actions and reducers, что дает flexibility выбирать разные архитектурные подходы.
    - можно мутировать store. **Immutability isn’t the preferred practice for using MobX.**
* **Apollo Client** – еще одна альтернатива **Redux**. Удобно при использовании в связке с **GraphQL**. Преимуществом **GraphQL** – когда мы делаем запрос на сервер to get data мы можем вызвать их в той форме, в которой хотим их получить. Можем выбрать необходимые данные из объекта и их форму.

**React recoil**

* **React recoil** – **React** library for **medium state** management. **Recoil** defines a **directed graph orthogonal to** but also **intrinsic** and **attached** to your **React** **tree**. **State** changes flow from the **roots** of this **graph** (which we call **atoms**) through **pure** **functions** (which we call **selectors**) and into components.

Использование:

1. Установить библиотеку: **npm install recoil**
2. Импортировать **RecoilRoot** и обернуть компоненту в **<RecoilRoot></RecoilRoot>**

import { RecoilRoot } from 'recoil';

import { CounterButton } from './CounterButton';

import './App.css';

const App = () => {

  return (

<RecoilRoot>

      <h1>State Management Example</h1>

      <CounterButton />

    </RecoilRoot>

  );

};

1. Создать recoil state/states с помощью atom.

import { atom } from 'recoil';

export const counterState = atom({

key: 'counterState',

default: 0,

});

import { atom } from 'recoil';

export const incrementByState = atom({

  key: 'incrementByState',

  default: 1,

});

**key** – string that Recoil uses to store this state behind the scenes. **default** – default value for state.

1. Использовать внутри компоненты. Что-среднее между **useState** and **useContext**.

import { useRecoilState } from 'recoil';

import { counterState } from './counterState';

import { incrementByState } from './incrementByState';

export const CounterButton = () => {

  const [numberOfClicks, setNumberOfClicks] = useRecoilState(counterState);

  const [incrementBy, setIncrementBy] = useRecoilState(incrementByState);

  return (

    <>

      <p>You have clicked the button {numberOfClicks} times.</p>

      <label>

        Increment By:

        <input

          value={incrementBy}

          onChange={(e) => setIncrementBy(Number(e.target.value))}

          type="number"

        />

      </label>

      <button onClick={() => setNumberOfClicks(numberOfClicks + incrementBy)}>

        Click

      </button>

    </>

  );

};

* **Recoil state** автоматически **шарится** **между компонентами** находящимися внутри **<RecoilRoot>.**
* В случае, когда не нужен setState из **useRecoilState**() то следует использовать другой хук – **useRecoilValue**()
* **Recoil selector –** they take the fundamental values that are expressed as atoms and they transform them in some way or combine them into another value.

**Routing**:

библиотека для комфортного роутинга на стороне фронта.

* + <**BrowserRouter**> - оборачивает, например **<App/>** компонент для начала использования
  + <**Routes**> - для группировки нескольких <Route/> в версии 4 для этого использовались <Switch>
  + <**Route** **path**='/' **element**={<**Component** />} /> - **path** - путь, в котором может быть **:id** и компонент, который будет отображен по этому пути.
  + <**Link** to='/' > import { **Link** } from **'react-router-dom'**; - используется для перехода с одного на другой **Route**
  + <**NavLink** **to**="/" **activeClassName**="**active**" /> - специальная версия <**Link** /> которая позволяет передавать активный класс, он сработает в случае совпадения **URL** с <**NavLink** **to**="/">
  + <**Outlet**> - для отображение вложенного Route внутри другого Route
    - <**Route path**='/**со-слешем**' element={< />} > <Route path=**'без-слеша'** element={< />} /> </Route>
    - **import** { **Outlet** } from **'react-router-dom'**; <div><**Outlet** /></div> в этом месте отобразится компонента, переданная во вложенный <**Route** />.
  + **useHistory**() - как и линк используется для перехода на другой Route:
    - **let history** = **useHistory**(); **history**.**push**(**новый-адрес**)
  + <**Redirect to='/путь-со-слешем'** > - используется рядом с **Route** для редиректов с одного URL на другой.
  + **useParams**() – позволяет получать данные из **URL**, которые находятся после /:**данные**/
    - **const** { **name** } = **useParams**();

**React Design Patterns**

эффективные решения для популярных проблем, возникающих при разработке. Улучшают производительность и простоту поддержки кода.

* + **регулярные проблемы,** которые решаются с помощью **design patterns:**
    - **Создание reusable layouts components(use styled-components) –** компоненты не должны знать где они будут отображены
      * **Split Screens –** базовая компонента, способнаяпринимать дочерние children. И, например занимаемые ими части экрана (flex: 1 и flex: 3)
        + <**SplitScreen** > <**LeftChildren** /><**RightChildren**/> </**SplitScreen**>
        + внутри: **SplitScreen** = ({ **children** }) const [**left**, **right**] = **children**; <>{**left**}</>
        + const **Styled** = **styled**.**div**` **flex**: ${(**props**) => **props**.**weight**}; `; - пример использования props в **styled**-**components**
      * **List and items. List** определяет только логику общего для **items** контейнера. Уникальная стилистика, обработка и вывод данных определяется внутри **ListItem.**
      * **Modals –** создается базовое универсальное модальное окно состоящие из компонент фона и тела. Внутри него размещаются {children} передаваемые внутрь Modal в качестве пропсов.
    - **Reusing сложной логики между компонентами**
* **Container Component –** компонент, который берет на себя загрузку данных и их последующее распределение между его {children} компонентами. Может использовать внутри useEffect с async функцией или принимать ее в виде пропсов. Возвращает

<>

{React.Children.map(children, (child) => {

if (React.isValidElement(child)) {

return React.cloneElement(child, { [resourceName]: state });

}

return child;

})}

</>

* **HOC** – **high order** **c**omponent. Вместо того чтобы просто возвращать **JSX**, возвращает **другой компонент,** в последующем возвращает **JSX**. По сути, **функция,** возвращающая компонент или **фабрика** для производства компонент, название функции начинается **с маленькой буквы**, потому что внутри них почти никогда не используется **JSX** напрямую. Может также принимать **другой** **компонент** через **props**. **HOC** оборачивает оригинальный компонент в контейнер посредством **композиции**. **HOC** является **чистой** **функцией** без **побочных** **эффектов**.
* **Применения HOC:** **добавление** дополнительной функциональности существующим компонентам; **Передача** поведения между несколькими компонентами. **Загрузка** данных. **Повторное использование** и **инкапсуляция** кода.
* **Custom Hooks –** специальные хуки которые комбинируют внутри себя стандартные хуки и способствуют **повторному** **использованию** кода, а также его **инкапсуляции**. **Must** start with **use**
  + - **Работа с формами** – **controlled** and **uncontrolled components**
* **Uncontrolled component –** следит за своим внутренним состоянием **самостоятельно** и высвобождает эти данные только после того как происходит какой-нибудь **event**, например **submit event** у формы. Использует **useRef**() хук и **ref={}** на **input** для сбора данных в момент нажатия **submit**. Предположительно вызывает меньше **lifecycle** функций из **updating** блока, что в свою очередь повышает производительность. Но поля можно **провалидировать** только после **submit**.
* **Controlled component** – контролируемая не следит за своим состоянием используя **useState**() и **onChange()** на **input**. **Controlled** компоненты предпочтительные так как они более **reusable**. Их также проще **тестировать** так как не нужно создавать **event**, чтобы посмотреть данные внутри. Доступна **валидация** прямо во время ввода значений в **inputs** – до нажатия **submit**. **Валидировать** поля удобно с помощью **useEffect**() в зависимостях [] кото нужно указать **inputName**.
  + - **Работа с модальными окнами** – могут быть **uncontrolled and controlled**. Оба получают {children} через пропсы и выводят их внутри себя через {children}. **Uncontrolled** сам содержит свой **state** и **процесит** открытие закрытие изнутри компоненты. Controlled modal контролируется снаружи и получает переменную **shouldShow** и функцию **onRequestClose()** которые контролируют модальное окно.
    - **Работа с onboarding flows (входящие потоки) –** это просто компонента, которая отображает разные шаги в **onboarding** процессе. Эта компонента пошагово опрашивает пользователя собирая данные в объект. Может быть **uncontrolled** and **controlled**. **Controlled** передает управление в родительскую компоненту, что позволяет отобразить дополнительные шаги или скрыть ненужные.
    - **Внедрение функциональных подходов**
* **PureComponent** изначально определяет функцию shouldComponentUpdate(nextProps, nextState), которая ответственна за принятие решения — нужно ли продолжать updating lifecycle или нет. Может значительно повысить **производительность**. В функциональной компоненте это можно реализовать через HOC.
* **useCallback –** хук позволяющий меморизировать **callback** и вызывать его повторно только при изменении **dependency array** **.**
* **Functional programming in React** – design pattern позволяет:
  + минимизировать мутации и изменения state в приложении
  + держать функции независимыми от внешних данных (**pure functions**)
  + функции как объекты первого класса
  + **controlled components** – функциональный подход. Мы можем получить компоненту в любом состоянии просто передавая ей необходимые **props**. Компонента не следить за своим state.
  + **functional components** – сам по себе способ написание поддерживает функциональное программирование.
  + **HOC** – high order components – функциональный подход описанный в концепции функциями первого класса.
  + **Recursive components** – компоненты которые используют рекурсию
  + **Component compositions =** это когда мы берем несколько разных компонент и объединяем их в одну. Своеобразный функциональный аналог наследования в ООП:

export const DangerButton = (props) => {

return <Button {...props} color="red" />;

};

* + **Partially applied components** – позволяют взять более базовые компоненты и путем передачи подмножества пропсов этих компонент создать более специфические компоненты. Следующий шаг в концепции композиций, через использование HOC.

const partiallyApply = (Component, partialProps) => {

return (props) => {

return <Component {...partialProps} {...props} />;

};

};

export const DangerButton = partiallyApply(Button, { color: 'red' });

**React performance**

* + использование **минифицированной** **production** сборки. **npm run build** чтобы создать production сборку работая через **react create app**.
  + Анализ производительности компонентов с помощью инструмента разработки «**Profiler**» в **Chrome dev tools.**
  + **Виртуализация** **длинных списков**: **react-window и react-virtualized** — это популярные библиотеки для оконного доступа. Они предоставляют несколько повторно используемых компонентов для отображения списков.
  + **Избежание согласования** - несмотря на то, что React обновляет только изменённые DOM-узлы, повторный рендеринг всё же занимает некоторое время. В большинстве случаев это не проблема, но если замедление заметно, то можно переопределив метод жизненного цикла shouldComponentUpdate. Если известны ситуации, в которых компонент не нуждается в обновлении, можно вернуть false из shouldComponentUpdate. Вместо того, чтобы писать **shouldComponentUpdate**() вручную, можно наследоваться от **React**.**PureComponent**. Это эквивалентно реализации **shouldComponentUpdate**() с **поверхностным** сравнением текущих и предыдущих пропсов и состояния.

|  |  |
| --- | --- |
| class CounterButton extends React.Component {  constructor(props) {  super(props);  this.state = {count: 1};  }  shouldComponentUpdate(nextProps, nextState) {  if (this.state.count !== nextState.count) {  return true;  }  return false;  }  render() {<></>}  } | class CounterButton extends React.PureComponent {  constructor(props) {  super(props);  this.state = {count: 1};  }  render() {<></>}  } |

* **избегать мутирования значений** - возвращать новый объект вместо того, чтобы мутировать исходный

function updateColorMap(colormap) { return {...colormap, right: 'blue'}; }

**Server-Side Rendering (SSR)**

* **Server-Side Rendering (SSR).** Нормальное поведение обычного React приложения – клиент делает запрос на сервер и загружает index.html который чаще всего не информативен и содержит только базовую html структуру. По сути он пуст для пользователя. Далее этот файл указывает браузеру загрузить React scripts, которые уже запустят рендеринг всех html элементов на странице. Однако с **Server-Side rendering** server – тот, кто заботится о запуске скриптов и рендеринге всех html элементов.
* **Сравнение Client-Side rendering и Server-Side Rendering:**

|  |  |
| --- | --- |
| **Client-Side Rendering** | **Server-Side Rendering** |
| Renders app to HTML in the user’s browser | Renders app to HTML on the server и просто шлет законченный HTML документ клиенту |
| **Процесс:**  Loads index.html from server,  loads JavaScript bundle from server,  runs bundle,  display app,  loads data | **Процесс:**  runs JavaScript bundle,  loads data (much easier because we are on the server)  creates HTML document,  send it to client side |
| **Pluses:**  Less нагрузка на the server | **Pluses:**  Faster user experience  Better for SEO |
| **Minuses:**  generally slower user experience | **Minuses:**  More strain on the server |

* **Basic Server-Side rendering express example:**

// Express

import express from 'express';

// React

import React from 'react';

import { renderToString } from 'react-dom/server';

// Components

import { Home } from './src/pages/Home';

const app = express();

const app = express();

app.use(express.static('./build', { index: false })); // Staticky serve the files inside build folder

// but don’t load base index.html by default

app.get('/\*', (req, res) => {

const reactApp = renderToString(<Home />);

return res.send(`

<html>

<body>

<div id="root">${reactApp}</div>

</body>

</html>

`);

});

app.listen(8080, console.log('Server is listening on port 8080'));

* **Server-Side rendering practice:** 
  + **Express**. Use **npx nodemon --exec npx babel-node server.js** command to run the **server** with **nodemon** and **babel**.
  + Use **ReactDOM.hydrate** вместо **ReactDOM.render** внутри **index.js**, что позволит добавлять **React** в получаемые с сервера **HTML:**
  + Также необходимо перенести <**BrowserRouter**> </**BrowserRouter**> обертку из **App** в **index**.**js** для нормальной работы приложения с **SSR:**

import React from 'react';

import ReactDOM from 'react-dom';

import { BrowserRouter } from 'react-router-dom';

import App from './App';

// hydrate takes pre-rendered HTML that we're going to get from the server and adds React to it

// Despite of we're getting an HTML from the server, we still want to behave and rerender like ReactApp

ReactDOM.hydrate(

  <React.StrictMode>

    <BrowserRouter>

      <App />

    </BrowserRouter>

  </React.StrictMode>,

  document.getElementById('root')

);

* + **Worked example of SSR server.js file:**

/ Express

import express from 'express';

import path from 'path';

import fs from 'fs';

// React

import React from 'react';

import { renderToString } from 'react-dom/server';

import { StaticRouter } from 'react-router-dom';

// Components

import App from './src/App';

const app = express();

// Staticky serve the files inside build folder but don’t load base index.html by default

app.use(express.static('./build', { index: false }));

app.get('/\*', (req, res) => {

  const reactApp = renderToString(

    <StaticRouter location={req.url}>

      <App />

    </StaticRouter>

  );

  const templateFile = path.resolve('./build/index.html');

  fs.readFile(templateFile, 'utf8', (err, data) => {

    if (err) {

      return res.status(500).send(err);

    }

    return res.send(

      data.replace('<div id="root"></div>', `<div id="root">${reactApp}</div>`)

    );

  });

});

app.listen(8080, () => {

  console.log('Server is listening on port 8080');

});

* + **styled components** don’t work **with SSR.** We need to add addition code to the server to make it work. And also don’t forget to add {{ styles }} to index.html для успешной замены: .replace('{{ styles }}', sheet.getStyleTags()).

// Styles

import { ServerStyleSheet } from 'styled-components';

…

app.get('/\*', (req, res) => {

  const sheet = new ServerStyleSheet();

  const reactApp = renderToString(

    sheet.collectStyles(

      <StaticRouter location={req.url}>

        <App />

      </StaticRouter>

    )

  );

…

return res.send(

      data

        .replace('<div id="root"></div>', `<div id="root">${reactApp}</div>`)

.replace('{{ styles }}', sheet.getStyleTags())

    );

…

* + Следует также помнить, что **window** и **document** не будут работать при **SSR**.

**Вопросы на интервью**

1. **Что такое React?**
   * Это JavaScript библиотека для создания пользовательских интерфейсов. Деклоративен – достаточно описать как части интерфейса выглядят в разных состояниях, что упрощает написание, отладку и делает код предсказуемым. Основан на инкапсулированных компонентах, которые как кирпичики используется при построении сложных пользовательских интерфейсов. В основе лежат такие понятия как виртуальныйDOM и JSX. Может работать на сервере (SSR, next.js) и на мобильных приложениях (React Native). Не полноценный фреймворк.
2. **Можно ли использовать React без webpack и create React app?**
   * Можно. Нужно подключить скрипты react, react-dom и js file с кодом в конец body. Чтобы иметь возможность использовать JSX нужно подключить еще и babel скрипт, добавив в скрипты с JSX type="text/babel". Следует помнить, что работа сайта при этом замедляется.

**JSX**

1. **Что такое JSX?**
   * JSX – Дополнение к синтаксису JavaScript. JS XML. Объединяет JS и HTML (логику и разметку) позволяя писать на HTML в React компонентах. Улучшает читабельность и производительность. Обязательно нужен babel или другой транспилятор. JSX использует стиль camelCase для свойств. Например, class становится className в JSX, а tabindex становится tabIndex.
2. **Можно ли использовать React без JSX?**
   * Да, JSX совсем необязателен, но значительно упрощает разработку и читабельность. Достаточно использовать JavaScript опираясь на React.createElement(). К слову, после транспиляции JSX представляет собой обычный JavaScript.
3. **Насколько безопасен ли JSX?**
   * Данные, введённые пользователем, можно безопасно использовать в JSX. По умолчанию React DOM экранирует все значения, включённые в JSX перед тем, как отрендерить их. Всё преобразуется в строчки, перед тем как быть отрендеренным. Это помогает предотвращать атаки межсайтовым скриптингом (XSS).

**Виртуальный дом (Virtual DOM).**

1. **Что такое Virtual DOM виртуальный дом и как он работает?**
   * Подход, при котором виртуальное представление дом хранится в памяти в виде JS объекта и этот виртуальный дом синхронизируется с настоящим, обновляя только те его части, где необходима повторная отрисовка. В React для этого используется библиотека React-DOM. Сам процесс называется **согласование(reconciliation).** Также React использует внутренние объекты называемые **волокна (fibers) –** js объекты, содержащие информацию об компоненте, входные параметры и результат.

**React элемент и React компонент**

1. **Что такое React элемент?**
   * JSX представляет собой объекты - Babel компилирует JSX в вызовы React.createElement(). React.createElement() проводит некоторые проверки и создаёт объект. Эти объекты называются React-элементами. Они содержат поля type (со значением например ‘h1’ или ‘div’), props и др. Элементы — мельчайшие кирпичики React-приложений.
2. **Можно ли мутировать** **React элемент?**
   * Элементы React иммутабельны. После создания элемента нельзя изменить его потомков или атрибуты. Элемент похож на кадр в фильме: он отражает состояние интерфейса в конкретный момент времени.
3. **Что такое React компонент?**
   * 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. Позволяют разбить интерфейс на независимые части, их можно складывать вместе и использовать несколько раз. Существуют функциональные и классовые компоненты.
4. **Что такое функциональный компонент?**
   * это компонент, описанный как функция, она получает данные в одном объекте («пропсы») в качестве параметра и возвращает React-элемент. Буквально являются функцией.
5. **Что такое классовый компонент?**
   * это компонент, описанный как ES6 класс, который наследуется от React.Component. Имеет обязательный метод render(). Определяет свой стейт и получает пропсы используя constructor(). Обращается к пропсам и стейту через this. Также возвращает React элемент.
6. **Чем отличаются функциональный компонент от классового?**
   * С точки зрения React, эти два компонента эквивалентны. Основное отличие заключается в синтаксисе.
7. **Как ты понимаешь выражение React компонент — это чистая функция?**
   * React-компоненты обязаны вести себя как чистые функции по отношению к своим пропсам. Это одно из основополагающих правил React, которое нельзя нарушать.

**Пропсы (props) и стейт (state)**

1. **Что такое пропсы (props)?**
   * props – свойства, данные, которые передаются от родительской к дочерней компоненте. Доступны только для чтения - компонент никогда не должен что-то записывать в свои пропсы — вне зависимости от того, функциональный он или классовый, так как компоненты – чистые функции и они не меняют свои входные данные. И не могут быть отправлены обратно – однопоточный поток данных.
2. **Что такое стейт (state)?**
   * Это внутренние состояние компоненты. Оно контролируется и доступно только конкретному компоненту. Объект, определяющий представление и поведение компоненты. Используется для представления динамических данных. Нельзя изменять напрямую. Нужно использовать setState().
3. **setState синхронный или асинхронный?**
   * Асинхронный. setState говорит React запустить следующую итерацию рендера, однако React может оптимизировать этот процесс и несколько вызовов setState приведут к одному рендеру.
4. **Что обозначает термин однонаправленный поток данных в React?**
   * Состояние компонент инкапсулировано, не важно функциональны они или классовые. Компонент может передать своё состояние вниз по дереву в виде пропсов дочерних компонентов. Но получив эти пропсы дочерний компонент не знает, откуда они взялись изначально — из состояния, пропсов или просто JavaScript-выражения.
   * Это, в общем, называется «нисходящим» («top-down») или «однонаправленным» («unidirectional») потоком данных. Состояние всегда принадлежит определённому компоненту, а любые производные этого состояния могут влиять только на компоненты, находящиеся «ниже» в дереве компонентов.

**Жизненный цикл (React Lifecycle)**

1. **Что такое жизненный цикл (Lifecycle) React компонента?**
   * Это период с момента первоначальный рендеринга компоненты (монтирования – mounting в DOM), до удаления DOM-узла, созданного компонентой (размонтирования – unmounting).
2. **Назови методы жизненного цикла (Lifecycle) компонента?**
   * **mounting** 
     + - initialization (initial state or default props)
       - componentWillMount() – перед рендерингом
       - render() - рендер
       - componentDidMount() – запускается после того, как компонент отрендерился в DOM
     + **updating** 
       - componentWillReceiveProps() – перед получением пропсов
       - shouldComponentUpdate() - true or false, можно принудительно передать true для обновления
       - componentWillUpdate() – перед обновлением
       - render() - рендер
       - componentDidUpdate() – после обновления
     + **unmounting** 
       - componentWillUnmount – после удаления компоненты. Для очистки памяти.

**Rendering и отрисовка (обновление DOM)**

1. **Что вызывает обновление rendering компонентов?**
   * setState у useState и useReducer – изменение ее состояния
   * rendering родительского компонента приведет к рендерингу дочернего через изменения props
   * в классовых компонентах есть функция forceUpdate()
2. **Что НЕ вызывает обновление rendering компонентов?**
   * useState и useReducer не приведут к обновлению если состояние не изменилось.
   * если мутировать состояние напрямую
3. **Как реализовать forceUpdate() в функциональной компоненте?**
   * const [\_, forceUpdate] = useReducer((x)=> x + 1, 0) – можно также обернуть это в custom hook
   * Это может быть полезно при использовании Object.assign(obj, {‘new-field’: ‘value’}), который быстрее чем деструктуризация ({…obj, new-field: value}) НО не вызывает ререндеринг
4. **Тоже ли самое rendering и отрисовка (обновление DOM)?**
   * НЕТ – Компонент может заререндериться и без визуальных изменений. Рендеринг родительской компоненты вызовет рекурсивно рендеринг дочерних и те дочерние, что вернули тот же самый результат, т. е. не изменились - не будут перерисованы в DOM. Но React должен сделать ререндер, чтобы определить эти различия.

**memo и useMemo**

1. **В чем разница между memo и useMemo?**
   * memo – это компонент высшего порядка, а useMemo – это хук возвращающий меморизированное значение функции. Оба нужны для повышения производительности. memo подходит для случаев, когда компонент рендерит одинаковый результат при одних и тех же значениях пропсов в этом случае результат будет меморизирован.
2. **Как работает и применяет memo?**
   * React будет использовать результат последнего рендера при одних и тех же значениях пропсов избегая повторного рендеренга. Достаточно обернуть компонент в memo импортированный из ‘react’: export const Component = memo(Component);. Мемо используется если в компоненте меняются только пропсы, если же в компоненте используется state – компонент будет повторно рендериться при изменении состояния или контекста. При использовании memo пропсы сравниваются поверхностно(при помощи оператора ===). Можно также передать вторым параметром колбек функцию, чтобы кастомизировать процесс сравнения пропсов.
3. **Как работает и применяет useMemo?**
   * Возвращает меморизированное значение функции, которая делает некоторые затратные вычисления. Эта оптимизация позволяет избежать дорогостоящих вычислений при каждом рендере. useMemo будет повторно вычислять меморизированное значение только тогда, когда будет изменено значение зависимости. const isPrime = useMemo( () => checkPrime(num), [num] )
4. **Что такое Pure Component?**
   * Компонент является чистым, если он гарантированно возвращает один и тот же результат при одинаковых пропсах и состоянии. Чистые компоненты имеют лучшую производительность за счет поверхностного сравнения пропсов и стейта. Для классового компонента есть метод **shouldComponentUpdate**() – это необязательный метод и если он вернет false, то React пропустит рендеринг компонента. Может содержать любую логику сравнения пропсов и стейта. Также есть class **PureComponent** он может быть использован (вместо обычного в связке **shouldComponentUpdate**)**.** Вместо этих штук в функциональных компонентах используются memo.

**Code splitting (разделение кода)**

1. **Что обозначает термин code splitting (разделение кода) в React?**
   * В первую очередь это обозначает что вместо отправления всего кода React приложения клиенту целиком, мы отправляем его по частям по мере необходимости. Это техника обеспечивает производительность и экономит ресурсы так как позволяет уменьшить величину кода, которую необходимо загрузить клиенту. Реализуется через { lazy, Suspense } from ‘react’. Реализация:

const One = lazy(() => import('./One'));

const Two = lazy(() => import('./Two'));

const Three = lazy(() => import('./Three'));

export const About = () => {

  return (

    <>

      <h1>About</h1>

      <Suspense fallback={<p>Loading Components...</p>}>

        <One />

        <Two />

        <Three />

      </Suspense>

    </>

  );

};

* + Следует помнить, что lazy работает только с экспортом по умолчанию (export default).
  + Компоненты внутри Suspense будут загружены в виде трех отдельных чанков (chunk) только после того, как пользователь перейдет на About page.
  + Также можно разделить код основываясь на Routes, (роутах). Ведь, по сути, мы не обязаны загружать код до того, как пользователь перейдет на определенную страницу.

1. **Как разделить код основываясь на Routes (роутах)?**
   * Достаточно обернуть в Suspense наши Routes (роуты) или Switch если используется 5 версия. И импортировать с помощью lazy() компоненты передаваемы в каждый Route (роут). При переходе на каждый из Route (роутов) будет подгружаться соответствующий chunk (чанк).
2. **Когда НЕ следует использовать разделение кода (code splitting)?**
   * Не имеет смысла использовать разделение кода при работе с небольшими компонентами. Также нет смысла использовать разделение кода внутри компоненты, которая уже была подвержена этому процессу ранее. Если только в ней нет дополнительной логики по скрытию отображению компонент внутри нее.

**Предохранители (компоненты Error Boundary)**

1. **Что такое предохранители (компоненты Error Boundary)?**
   * 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. Предохранители отлавливают ошибки при рендеринге, в методах жизненного цикла и конструкторах деревьев компонентов, расположенных под ними.
   * Предохранители работают как JavaScript-блоки catch {}, но только для компонентов.
   * Классовый компонент является предохранителем, если он включает хотя бы один из следующих методов жизненного цикла: static getDerivedStateFromError() или componentDidCatch()
   * Только классовые компоненты могут выступать в роли предохранителей
2. **В каких случаях** **предохранители (компоненты Error Boundary) не сработают(не поймают ошибки)?**
   * в обработчиках событий;
   * в асинхронном коде (например колбэках из setTimeout или requestAnimationFrame);
   * в серверном рендеринге (Server-side rendering);
   * в самом предохранителе (а не в его дочерних компонентах).
3. **Напишите предохранитель (компонент Error Boundary).**

export default class ErrorBoundary extends React.Component {

  constructor(props) {

    super(props);

    this.state = { hasError: false };

  }

  static getDerivedStateFromError(error) {

    // Обновить состояние с тем, чтобы следующий рендер показал запасной UI.

    return { hasError: true };

  }

  componentDidCatch(error, errorInfo) {

    // Можно также сохранить информацию об ошибке в соответствующую службу журнала ошибок

    logErrorToMyService(error, errorInfo);

  }

  render() {

    if (this.state.hasError) {

      // Можно отрендерить запасной UI произвольного вида

      return <h1>Что-то пошло не так.</h1>;

    }

    return this.props.children;

  }

}

1. **Где стоит размещать предохранитель (Error Boundary)?**
   * Степень охвата кода предохранителями остаётся на усмотрение разработчиков. Можно защитить им навигационные (route) компоненты верхнего уровня, чтобы выводить пользователю сообщение «Что-то пошло не так», как это часто делают при обработке ошибок серверные фреймворки. Или можно охватить индивидуальными предохранителями отдельные виджеты, чтобы помешать им уронить всё приложение.
   * Более детальный охват предохранителями обеспечит лучший опыт взаимодействия с приложением при возникновении ошибок.
2. **Что будет если ошибка не будет отловлена не одним из предохранителей (Error Boundary)?**
   * Начиная с React 16, ошибки, не отловленные ни одним из предохранителей, будут приводить к размонтированию всего дерева компонентов React.
   * Хотя принятие этого решения и вызвало споры, бо́льшим злом будет вывести некорректный UI, чем удалить его целиком. К примеру, в приложении типа Messenger, вывод поломанного UI может привести к тому, что пользователь отправит сообщение не тому адресату. Аналогично, будет хуже, если приложение для проведения платежей выведет пользователю неправильную сумму платежа, чем если оно не выведет вообще ничего.