**React** – **JS** библиотека. Не полноценный фреймворк. Использует компонентный подход, модульность и повторное использование компонент. **Virtual DOM, JSX**. Может выполнять server-side rendering. Тестирование проще.

* **Virtual DOM** – легкий **JS** объект. Копия реального **DOM**. Дерево с элементами в качестве объектов с данными. Дерево обновляется в ответ на изменения стейта, пропсов, добавления и удаления узлов. Производительность.
* **Virtual DOM PROCESS** – **Virtual DOM** обновляется при изменениях. Вычисляется разница между новым Virtual и Real DOM. Обновляет только измененную часть Real DOM. Производительность.
* **JSX – JS XML**. Объединение **JS** и **HTML**. Улучшает читабельность и производительность. Обязательно нужен babel.
* **Синтаксис** **Class, Function:**

|  |  |
| --- | --- |
| Class ClassComponent extends React.Component {  constructor() {  super();  this.state = {name: ‘world’};  }  render () {  return (<h1>Hello {this.props.name} {this.state.secondName}<h1>)  } | const FuncComponent = ({name}) => {  const [name, setName] = useState(‘world’);  return (<h1>Hello {name} {secondName}<h1>)  } |

* **render()** – возвращает **React** элемент, который является компонентом DOM.
* **props** – свойства, данные, передаются от родительской к дочерней компоненте. Доступны только для чтения. И не могут быть отправлены обратно – однопоточный поток данных.
* **state** – состояние компоненты. Объект, определяющий представление и поведение компоненты. Используется для представления динамических данных.
* **Life cycle** – **mounting**, **updating**, **unmounting:**
  + **mounting** 
    - initialization (initial state or default props)
    - componentWillMount() – перед рендерингом
    - render() - рендер
    - componentDidMount() – после рендеринга
  + **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 сноса компоненты. Для очистки памяти.
* **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
* **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 –** библиотека для **state** менеджмента, используется для контроля состояния всего **JS** приложения основана на архитектуре **Flux**.
  + 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.
* **useSelector**((**state**) => **state**.**someReducer**.**someValue**) – импортируемый из react-redux хук для получения необходимых кверей из store.
* **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**

* **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));**
* **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.
* **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. Улучгают 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'}; }

* **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 -** 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).
  + import { **createContext** } from 'react'; const SomeContext = **createContext**(); - создание глобального контекста
  + <**Context.Provider** **value**={{ someValue, orSomeObj }}> - оборачивает компонент, например <App/>
  + import { **useContext** } from 'react'; const { someValue, orSomeObj } = **useContext**(SomeContext); - использование
* **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');

});