ЛАБОРАТОРНАЯ РАБОТА №6
REACTJS

Цель работы: получить навык разработки веб-приложений с помощью библиотеки ReactJS.

Задачи:

1. Изучить принципы работы с виртуальным DOM
2. Изучить основы JSX
3. Разработать 2 SPA (Single Page Application) с помощью библиотеки ReactJS: используя классовые и функциональные компоненты.

Результатами работы являются:

1. Разработанные сайты.
2. Подготовленный отчет.

REACTJS

ReactJS – JS-библиотека для создания пользовательских интерфейсов. Ключевые возможности библиотеки – работа с виртуальным DOM и использование декларативного HTML-подобного JSX-синтаксиса создания элементов интерфейса, преобразующегося в JS-скрипты, рисующие элементы в DOM (определена большая часть тегов HTML, также возможности JSX можно расширять собственными компонентами).

В самомо простом варианте использования, достаточно добавить библиотеку в зависимости проекта, а после в JS создать React-элементы и дорисовать их в виртуальный DOM к конкретному узлу с помощью метода ReactDOM.render(element, parentElement):

```
const myelement = <h1>I Love JSX!</h1>;
ReactDOM.render(myelement,
    document.getElementById('root'));
```

Однако, при работе над сложным проектом, состоящим из множества компонентов и большого числа внешних зависимостей проще использовать утилиту create-react-app (можно установить, используя менеджер пакетов npm, и запустить с помощью npx), создающую проект по умолчанию с удобным распределением файлов по директориям: в проекте содержится файл index.html, содержащий контент базовой страницы, содержащий корневой элемент; различные файлы ресурсов; JS-файлы с описание React-элементов; файл с перечнем необходимых зависимостей; а также файл описания действий жизненного цикла приложения и некоторых других дополнительных файлов. При разработке удобно использовать встроенный в node.js web-сервер express, автоматически обновляющий файлы проекта при изменении.

Есть 2 способа создания react-компонентов: на основе классов и функций. Рассмотрим их подробнее.

СОЗДАНИЕ КОМПОНЕНТОВ НА ОСНОВЕ КЛАССОВ

React-компоненты на основе классов представляют собой обычные JS-классы, наследуемые от класса React.Component и обязательно имеющие метод render(), возвращающий JSX-представление компонента (обязательно должен быть только один JSX-элемент верхнего уровня. При необходимости можно использовать родительские или <Reac.Fragment> если не желательно создавать новые элементы в DOM. Для подстановки результатов вычислений необходимо использовать {}). В данном примере создается компонент Car, затем используемый в другом компоненте Garage:

```
class Car extends React.Component {
    render() {
          return <h2>Hi, I am a Car!</h2>;
    }
}

class Garage extends React.Component {
    render() {
          return (
                <div>
                      <h1>Who lives in my Garage?</h1>
                      <Car />
                </div>
          );
    }
}
```

Часто для корректного поведения компонента бывает необходимо передать ему какие-то входные параметры (например, имя пользователя, или его роль). Для этого необходимо использовать специальный объект props: prop'ы передаются в JSX аналогично аттрибутам в HTML – указывается имя prop'а и его значение – в самом компоненте к данным props можно обратиться с помощью словаря this.props (обратите внимание! В методах React.Component, таких как render, контекст устанавливается в конструкторе React.Component. Для

этого в собственном компоненте необходимо создать конструктор, вызывающий конструктор базового класса, для каких-либо других методов контекст необходимо задавать). Изменние props приведет к созданию нового элемента (см. методы жизненного цикла ниже). Важно: при создании списков для идентификации элементов каждому элементу списка необходимо передать prop key, представляющий собой уникальный идентификатор.

```
class Car extends React.Component {
    constructor(props) {
          super(props);
    }
    render() {
          return <h2>I am a {this.props.brand}!</h1>;
    }
}

//использование компонента
const myelement = <Car brand="Ford" />;
```

Помимо входных параметров, компонент может обладать своим внутренним состоянием, не зависящим (по крайней мере, напрямую) от других компонентов (например, компонент авторизации может хранить имя пользователя, его возраст и т.д.). Для хранения этих данных используется объект state. Объект state инициализируется в конструкторе. Обратиться к объекту state в любом месте компонента можно используя синтаксис: this.state.propertyname. Чтобы изменить значение в объекте состояния, используйте метод this.setState() – никогда не мутируйте this.state напрямую, так как более поздний вызов setState() может перезаписать эту мутацию! Когда объект state изменяется, компонент ререндерится.

```
class Counter extends React.Component {
    constructor(props) {
          super(props);
          this.state = {
                count: 0
          }
    }
```

```
    render() {
          return <React.Fragment>
                <h2>Counter =
                      {this.state.counter}</h1>;
                <button onClick={() =>
                      this.setState({count:
                            this.state.count++})}>
                      Increment
                </button>
          <React.Fragment>
    }
}
```

Состояние используется также для создания управляемых компонентов – компонентов, хранящих свои данные не в себе, а в родительском react-компоненте (например, значение input'а можно хранить не в самом input'е, а в компоненте-форме, что позволит централизованно обрабатывать данные формы, например, проводить валидацию формы).

Каждый элемент в приложении «проживает» определенные стадии жизни, в reactJS возможно использовать специальные методы, так называемые методы жизненного цикла, для того чтобы в нужные моменты жизни элемента выполнять определенные действия (например, выполнить AJAX-запрос для получения данных, которые потом будут отображены в элементе). В начале элемента не существет в приложении, затем он создается (вызывается его конструктор), вызывается метод render, элемент монтируется в DOM (после чето вызывается метод componentDidMount), в какой-то момент элемент может изменить свое состояние (вызывается метод componentDidUpdate), а когда-то он может быть демонтирован из DOM (перед этим вызовется componentWillUnmount). Схема представлена на рис. 5.1. Это наиболее используемые методы жизненного цикла, но они представляют не полный перечень методов жизненного цикла.

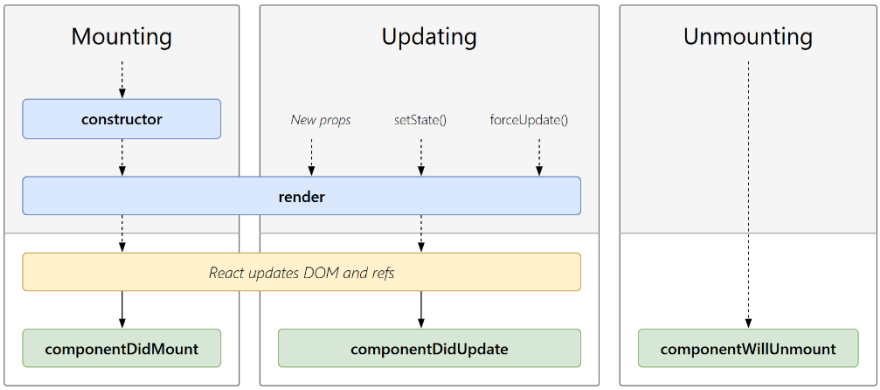

Рис. 5.1

Еще одной интересной возможностью reactJS являются рефы (ref). Рефы дают возможность получить доступ к DOM-узлам или React-элементам, созданным в рендер-методе. По сути реф – это переменная, которая хранит ссылку на react-элемент. Ситуации, в которых использование рефов является оправданным:

- Управление фокусом, выделение текста или воспроизведение медиа.
- Императивный вызов анимаций.
- Интеграция со сторонними DOM-библиотеками.

Пример использования рефов:

```
class MyComponent extends React.Component {
    constructor(props) {
          super(props);
          this.myRef = React.createRef();
    }
    render() {
          return <div ref={this.myRef} />;   }
    }

//установка фокуса в произвольном месте кода
this.myRef.current.focus();
```

СОЗДАНИЕ ФУНКЦИОНАЛЬНЫХ КОМПОНЕНТОВ

Изначально функциональные компоненты были облегченной версией создания компонентов – функциональные компоненты обладают более простым синтаксисом, при этом не обладают состоянием и не имеют методов жизненного цикла – однако, в настоящий момент с расширением функционала с помощью различных хук, являются рекомендуемым способом создания react-компонентов, т.к. использование собственных хук позволяет существенно понизить дублирование кода по сравнению с компонентами на основе классов.

Функциональный компонент представляет собой обычную JS-функцию, возвращающую JSX, в качестве входного параметра передаются props:

```
function Car(props){
    return <h2>I am a {props.brand}!</h1>;
}
```

Хуки — это функции, с помощью которых можно использовать состояние и методы жизненного цикла React из функциональных компонентов. Хуки не работают внутри классов — они дают возможность использовать React без классов.

Правила использования:

- Хуки следует вызывать только на верхнем уровне. Не вызывайте хуки внутри циклов, условий или вложенных функций (т.к. может нарушиться порядок вызова хук, что приведет к печальным последствиям).
- Хуки следует вызывать только из функциональных компонентов React. Не вызывайте хуки из обычных JavaScript-функций (опять же, порядок вызова хук станет непредсказуемым). Есть только одно исключение, откуда можно вызывать хуки — это пользовательские хуки.

Рассмотрим базовые хуки.

1. Хука состояния.

Вызов useState возвращает две вещи: текущее значение состояния и функцию для его обновления. Единственный аргумент useState — это начальное состояние: const [count, setCount] = useState(0);

Обращение к состоянию: {count}

2. Хука состояния часто используется вместе с хукой эффекта (загрузка и сохранение данных).

Хук эффекта даёт возможность выполнять побочные эффекты в функциональном компоненте. Побочными эффектами в React-компонентах могут быть: загрузка данных, оформление подписки и изменение DOM вручную (хук useEffect представляет собой совокупность методов componentDidMount, componentDidUpdate, и componentWillUnmount):

```
useEffect(() => { document.title =
    `Вы нажали ${count} раз`;  },
    [count]
);
```

Вторым необязательным параметром передается массив переменных, на измении которых должен срабатывать эффект. Если эффект возвращает функцию, React выполнит её только тогда, когда наступит время сбросить эффект. Пример использования хук:

```
function Garage(props) => {
    const [cars, setCars] = useState([])
    useEffect(() => {
                let result = await fetch(…)
                if (result.ok){
                      setCars(result.json())
                }
          }
    )

    return <React.Fragment>
          {
                cars.map(item => <h2>{item.name}<h2>)
          }
    </React.Fragment>
}
```

3. Для работы с рефами в функциональных компонентов используется хука useRef. useRef возвращает изменяемый ref-объект, свойство .current которого инициализируется переданным аргументом (initialValue). Возвращённый объект будет сохраняться в течение всего времени жизни компонента. Пример использования:

```
function TextInputWithFocusButton() {
    const inputEl = useRef(null);
    const onButtonClick = () => {
          //`current` указывает на элемент `input`
          inputEl.current.focus();
    };
    return (
          <React.Fragment>
                <input ref={inputEl} type="text" />
                <button onClick={onButtonClick}>
                      Установить фокус на поле ввода
                </button>
          </React.Fragment>
    );
}
```

Но хук useRef() полезен не только установкой атрибута с рефом. Он удобен для сохранения любого мутируемого значения, по аналогии с тем, как используются поля экземпляра в классах.

Это возможно, поскольку useRef() создаёт обычный JavaScript-объект. Единственная разница между useRef() и просто созданием самого объекта {current: ...} — это то, что хук useRef даст один и тот же объект с рефом при каждом рендере. Мутирование свойства .current не вызывает повторный рендер.

Перечисленные хуки дают базовые представления о работе с функциональными хуками, другие полезные хуки рассмотрены в документации https://ru.reactjs.org/docs/hooks-reference.html.

Кроме базовых хук у прикладного программиста есть возможность создавать свои собственные кастомные хуки, комбинируя существующие. Созданные хуки можно многократно использовать в различных компонентах, что позволяет сократить дублирование кода

(в компонентах на основе классов пришлось бы писать одну и ту же логику для каждого из компонента). Пример пользовательской хуки:

```
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;
}
```

Данная хука при монтировании компонента подписывается на некое API чата, через которое будет получен статус друга (онлайн или оффлайн) с указанным идентификатором, по демонтировании компонента произойдет отписка от API чата и возвращает перменную, содержащую значение статуса друга. Т.е. данная хука комбинирует хуку состояния и хуку эффекта. В дальнейшем может многократное использование данной хуки в разных компонентах, например, в компоненте, отображающем список друзей (классические зеленые и красные «точки» на иконках друзей) и в компоненте диалога с другом (надпись Online или Offline). Используя компоненты на основе классов, пришлось реализовывать одну и туже логику в обоих компонентах.

Благодаря тому, что есть возможность создавать пользовательскте хуки, помимо стандартных хук react’a, существует большое количество хук, реализованных в различных библиотеках, таких, как, например, Formik.

ЗАДАНИЕ НА ЛАБОРАТОРНУЮ РАБОТУ

Разработать 2 сайта тематики, согласно полученному варианту (используя react-компоненты на основе классов и функций).

Сайт должен состоять минимум из 2-х частей с возможностью перехода между ними (заданием переменной в state элемента).

Обязательно использование вложенных React-элементов (с передачей props, например, введенных в родительском элементе данных), условного рендеринга.

Использование готовых компонентов и сторонних библиотек запрещено.

ВАРИАНТЫ ЗАДАНИЙ

1. Сайт школы
2. Сайт фитнес-центра
3. Новостной сайт
4. Метеорологический сайт
5. Сайт музыкального исполнителя
6. Сайт кинотеатра
7. Сайт отеля
8. Сайт аэропорта
9. Сайт телеканала
10. Сайт радиостанции
11. Сайт автосалона
12. Сайт ресторана
13. Сайт университета
14. Сайт библиотеки
15. Сайт театра
16. Сайт ветеренарной клиники
17. Сайт туристической фирмы
18. Сайт агентства по продаже недвижимости
19. Сайт букмекерской компании
20. Сайт биржи криптовалют

КОНТРОЛЬНЫЕ ВОПРОСЫ И ЗАДАНИЯ

1. Охарактеризуйте JSX.
2. Приведите пример кода рендера react-компонента к какому-либо узлу DOM.
3. Охарактеризуйте props.
4. Охарактеризуйте state.
5. Приведите методы жизненного цикла react-компонента.
6. Опишите маханизм использования рефов.
7. Сравните react-компоненты на основе классов и на основе функций.
8. Перечислите ограничения, накладываемые на использование хук.
9. Опишите useState.
10. Опишите useEffect.
11. Опишите useRef.
12. Приведите пример создания собственной хуки.

ФОРМА ОТЧЕТА ПО ЛАБОРАТОРНОЙ РАБОТЕ

На выполнение лабораторной работы отводится 7 часов: 6 часов на выполнение работы, 1 час на подготовку и защиту отчета по лабораторной работе.

Структура отчета: титульный лист, цель и задачи, формулировка задания (вариант), этапы выполнения работы, исходный код разработанного сайта, результаты выполнения работы (скриншоты), выводы.