

Министерство науки и высшего образования Российской Федерации
Федеральное государственное бюджетное образовательное
учреждение высшего образования
**ИРКУТСКИЙ НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ
ТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ**
Институт заочно-вечернего обучения

Допускаю к защите

Руководитель Ю. Р. Басиров
подпись, И. О. Фамилия

Разработка элементов Front-end для веб-сервиса

Наименование темы

ПОЯСНИТЕЛЬНАЯ ЗАПИСКА
к курсовому проекту по дисциплине
«Технология программирования»

1.011.00.00 ПЗ

обозначение документа

Разработал студент группы ЭВМбз-16-1

подпись

А. А. Михиденко

И. О. Фамилия

Нормоконтроль

подпись

Ю. Р. Басиров

И. О. Фамилия

Курсовой проект защищена с оценкой _____

Иркутск 2021

Министерство науки и высшего Российской Федерации
Федеральное государственное бюджетное образовательное
учреждение высшего образования
**ИРКУТСКИЙ НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ
ТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ**
Институт заочно-вечернего обучения

**ЗАДАНИЕ
НА КУРСОВОЙ ПРОЕКТ**

По курсу: Технология программирования
Студенту: Михиденко Алексею Анатольевичу
Тема проекта: Разработка элементов Front-end для веб-сервиса

Исходные данные.

Изучить принципы технологии SPA. Подобрать и изучить материал по JS библиотеке React. Имея полученные данные, написать пользовательский интерфейс для абстрактного back-end. В работе должна быть отображена логика взаимодействия интерфейса с основными элементами HTML разметки - DOM элементами.

Рекомендуемая литература:

1. Сайт о программировании. [Электронный ресурс] : Руководство по React. <https://metanit.com/web/react/> (дата обращения 31.03.2021).
2. Официальная документация по React. [Электронный ресурс] : Основные понятия React. <https://ru.reactjs.org/docs/getting-started.html> (дата обращения 31.03.2021).

Графическая часть на 15 листах

Дата выдачи задания «31» марта 2021 г.

Задание получил _____ А. А. Михиденко

Дата представления работы руководителю «30» Марта 2021 г.

Руководитель курсового проекта: _____ Ю. Р. Басиров

Содержание

Введение.....	4
1 Теоретический материал.....	5
2 Постановка задачи.....	18
3 Выполнение задания.....	18
Заключение.....	24
Список использованных источников.....	25

Введение

На сегодня и в будущем развитая индустрия ИТ позволяет человеку реализовывать самые необычные идеи, которые интерпретируются нами всеми возможными методами передачи информации и которые явно зависят от состояния общей индустрии, в особенности технологической.

Одни из таких методов - это метод визуализации элементов информации, которые является по нашему мнению ключевыми. Этими элементами могут быть какие угодно данные подпадающие под абстрактное представление понятное человеку: символы, схемы, фигуры, и пр. Все это получает наибольший эффект когда такого рода информация приобретает разнообразное представление: имеет цвет, форму, присутствие элементов составляющих анимацию, а события управления тем или иным элементом или его свойством только улучшает восприятие и ключевые показатели взаимодействия человека и ЭВМ. Ведь в конечном итоге, мы люди заставляем решать эти сложные и рутинные задачи - вычислительные комплексы, которые основаны на непонятных для обычного обывателя вещах. Даже для самого продвинутого и обученного современными знаниями человека, так же необходима составляющая, которая будет не усложнять его деятельности, а наоборот упрощать. Такими могут быть ученые, врачи, хирурги, кассиры, водители транспорта или космонавты, и на самом деле, этот список не имеет границ.

Так что, одним из таким инструментов на сегодняшний день может по праву является архитектура SPA, а так же те технологии, которые ложатся в основу работы этой архитектуры. В данном случае речь идет о веб-сервисах, а точнее о пользовательском интерфейсе, который как раз и является своеобразным мостом между человеком, ЭВМ и данными. Которыми, по праву оперируют обе стороны.

1 Теоретический материал

Одностраничное приложение, (SPA) - веб-приложение(сайт), использующий в своем арсенале всего лишь один HTML документ, как оболочку для всех возможных вариантов интерпретации. Взаимодействие с пользователем достигается за счет динамических изменений структуры этого документа. Основным достоинством такого вида приложения является тот факт, что такая страница загружается всего лишь один раз, и при взаимодействии с пользователем - не требует, в большинстве случаев, перезагрузки контента.

Снижение задержек при передаче данных, быстрый отклик на действие пользователей - это главный двигатель прогресса в этой технологии. И одним из таких ярких примеров является JS библиотека React.

Виртуальный DOM - точная копия объектной модели документа в интерпретации JS. Данная абстракция применяется в React [1], что обеспечивает высокую производительность при взаимодействии между JavaScript. Поскольку для изменения чего либо, сначала необходимо прочитать DOM модель файла, затем обработать результат и снова прочитать модель для изменения того или иного компонента. На эту операцию затрачивается время, которое критично влияет на производительность JavaScript логики.

JavaScript XML, (JSX) - инструмент для создания элементов React. Удобное представление функциональной логики и объектов в одном документе, по принципу разделения ответственности, посредством комбинации кода JS и разметки XML.

После компиляции, все выражения JSX превращаются в обычный вызов функций JS и вычисляются в объекты JS. Так же, важной особенностью является тот факт что все DOM React элементы в JSX экранируются, что позволяет избегать XSS атак. Фича является рекомендуемым способом создания интерфейсов.

ОСНОВЫ JSX

```
<script type="text/babel">
  const user = {
    id : 5,
    age: 33,
    firstName: 'Tom',
    lastName: 'Smit',
    getFullName: function(){return `${this.firstName} ${this.lastName}`;}
  };
  ReactDOM.render(
    <div id={user.id}>
      <p>Полное имя: {user.getFullName()}</p>
      <p>Возраст: {user.age}</p>
    </div>,
    document.getElementById("app")
  )
</script>
```

Для определения, что код должен иметь принадлежность к React JSX, необходимо для элемента script указать свойство type="text/babel".

Как говорилось ранее, JSX после процесса компиляции(babel) представляется объектом JS, для вызова которых можно применять {...} скобки.

За рендер данных, в React DOM отвечает функция *ReactDOM.render*, которая первым аргументом принимает в виде простого текстового формата(в данном случае HTML разметку), и идентификатор DOM-элемента в структуре(DOM) документа.

Фигурные скобки {}, являются встраиваемыми выражениями Native JS, в JSX. Однако, React имеет внутренний транслятор, который позволяет распознавать не только простые JS выражения, а так же интерпретировать свою абстрактную модель, которая построена поверх Native JS.

Т.е. ряд ключевых особенностей, которые необходимо учитывать при работе с DOM элементами страницы при помощи JSX. Например тот факт что JSX работает исключительно с константами, т.е. для того что бы записать банальное свойство типа *class=name* для любого HTML элемента. В нотации JSX это будет выглядеть как {*className*}, где *className* должен обязательно быть объявленным как константа, которая содержит в себе строку, имя класса CSS.

Это самый простой способ продемонстрировать как работает JSX и React.

Для сравнения посмотрим на следующий код:

```
<div className={userClassName} style={{color: 'red', fontFamily: 'Verdana'}}>
```

который по сути идентичен этому:

```
<div className={userClassName} style={styleObj}>
```

при объявленной константе:

```
const styleObj = {  
  color: 'red',  
  fontFamily: 'Verdana'  
};
```

Компилятор Babel

Данная JS библиотека [2] содержит в своем арсенале реализацию многих возможностей новых стандартов ES2015+, которые на данный момент поддерживаются не всем браузерами.

Для React это означает что разработчик получает ряд новых возможностей, такие например как создание отдельных классов для своих компонентов, что очень упрощает представление объектной модели приложения.

Рассмотрим пример:

```
<script type="text/babel">  
  class Hello extends React.Component {  
    render() {  
      return <h1>Hello, React</h1>;  
    }  
  }  
  ReactDOM.render(  
    <Hello />,  
    document.getElementById("app")  
  )  
</script>
```

Элемент *Hello* вынесен в отдельный класс, который наследуется от класса *Component*. Эта нотация позволяет развивать компонент отдельно от другого кода, что очень, ну прям очень, соответствует ООП парадигме.

Теперь вспомним о принципе разделения, и мы знаем что JSX позволяет публиковать свои нотации в отдельных файлах. Для этого необходимо в секции `<script>` указать путь к файлу с расширением `*.jsx`.

Компонент (component)

Аналогичны функция JS. Существует три способа объявления:

- функциональный метод объявления(обычный, стрелочный)
- метод объявления классов

Стоит учитывать что названия классов, для применения их при вызове компонентов, необходимо публиковать начиная с символа в верхнем регистре. См. пример выше.

Свойства (props)

Это представление коллекции значений, которые ассоциируются с компонентами. Что позволяет создавать компоненты с динамическим содержимым.

Функциональный подход:

```
function Hello(props) {  
  return <div>  
    <p>Имя: {props.name}</p>  
    <p>Возраст: {props.age}</p>  
  </div>;  
}
```

Использование классов ES6:

```
class Hello extends React.Component {  
  render() {  
    return <div>  
      <p>Имя: {this.props.name}</p>  
      <p>Возраст: {this.props.age}</p>  
    </div>;  
  }  
}
```

Разница между этими двумя методами заключается в том, что при функциональном подходе, свойство передается в функцию как параметр, в которой инкапсулируются данные свойства.

При использовании класса, вызов которого сопровождается передачей данных в конструктор указанных элементов. Например:

```
<Hello name="Tom" age="33" />
```


defaultProps - идентификатор, который позволяет устанавливать элементам свойства параметры по умолчанию. Пример:

```
Hello.defaultProps = {name: "Tom", age: 22};
```

Для **обновление свойств** применяется объект ***state***.

События (events)

Основные отличия от обработки обычной обработки элементов JS:

- События в React используют *camelCase*
- В JSX в обработчик события *передается фун-я компонента*, а не строка

Самым распространенным подходом является установка привязки события в конструкторе класса компонента. Пример:

```
class ClickButton extends React.Component {
  constructor(props) {
    super(props);
    this.press = this.press.bind(this);
  }
  press(){
    console.log(this);
    alert("Hello React!")
  }
  render() {
    return <button onClick={this.press}>Click</button>;
  }
}
```

В этой JSX нотации используется событие нажатия кнопки, которое задается через *атрибут onClick()*. Не очень удобный способ, не правда ли?

Дело в том, что при таком вызове стоит учитывать особенности классовой модели React. Именно тот факт, что предварительно в конструкторе компонента необходимо вызвать конструктор базового класса, затем объявить свойство, которое будет ссылаться на функцию определенную обработчиком события. Сложность, компенсируется тем фактом, что вызов из формы представлен минимальной реализацией, где просто вызывается свойство класса.

Однако, есть еще метод передачи в качестве фактического параметра вызова обработчика лямбда функции.

В этом случае код будет выглядеть так:

```
class ClickButton extends React.Component {
  press(){
    console.log(this);
    alert("Hello React!");
  }
  render() {
    return <button onClick={() => this.press()}>Click</button>;
  }
}
```

Не трудно заметить что реализация класса значительно упростилась, так как теперь наш вызов позволяет получить прямой доступ к методу класса!

Золотой серединой в этом деле является упрощенный вызов формы и использование стрелочной функции. По существу которая функционирует следующим образом: объявляет публичное свойство-ссылку на метод выраженный в лямба нотации. Пример:

```
class ClickButton extends React.Component {
  press = () => {
    console.log(this);
    alert("Hello React!")
  }
  render() {
    return <button onClick={this.press}>Click</button>;
  }
}
```

Получение информации о событии. Формальный параметр (*e*) для любого метода обработчика - служебная информация, в которой хранится состояние события.

Передача параметров в обработчик события. Если есть необходимость в передаче формальных параметров в обработчик события, тогда в этом случае обработчик вызывается в нотации стрелочной функции.

```
<button onClick={() => this.print("Bob", 23)}>Print Bob</button>
```

Состояние (state)

Объект свойств описывает внутреннее состояние объекта и определяется только внутри компонента, а так же доступно только из компонента.

Объявление состояния происходит в конструкторе класса. Пример:

```
this.state = {welcome: "Добро пожаловать на сайт!"};
```

пример вызова(такой же как и у свойств):

```
{this.state.welcome}
```

Обновление состояния. Для обновления вызывается метод `setState()`;

```
this.setState({welcome: "Привет React"});
```

Формы(forms)

Работа с формами. React поддерживает формы, и благодаря своему функционалу предоставляет дополнительные возможности для работы с ними.

Рассмотрим фрагмент следующего кода:

```
render() {  
  return (  
    <form onSubmit={this.handleSubmit}>  
      <p><input type="text"  
        value={this.state.name}  
        onChange={this.onChange}/>  
      </p>  
      <input type="submit" value="Отправить" />  
    </form>  
  );  
}
```

В данном примере `onChange` и `handleSubmit` это служебные методы класса, которые являются обработчиками. В первом случае - нажатия кнопки, во втором - изменения объекта состояния класса.

Валидация форм. Процесс валидации позволяет проводить предварительное сопоставление формата данных с форматом указанного шаблона. Это предотвращает недопустимые изменения в объекте состояния класса.

Ссылки(Refs)

Ссылки применяются к любым DOM элементам. После установки атрибута для элемента, мы сможем сослаться на него. Для создания ссылок применяется метод `React.createRef()`.

В данном случае объявление происходит в конструкторе:

```
<input ref={this.nameField} />
```

Объявление ссылки происходит в конструкторе класса:

```
this.nameField = React.createRef();  
...  
var name = this.nameField.current.value;
```

Маршрутизация

Определение маршрутов. В React имеется своя система маршрутизации, которая позволяет сопоставлять запросы к приложению с определенными компонентами.

Для работы в браузере нам надо использовать модуль `react-router-dom` и так же будет необходимо получить ряд объектов:

```
const Router = ReactRouterDOM.BrowserRouter;  
const Route = ReactRouterDOM.Route;  
const Switch = ReactRouterDOM.Switch;
```

В этом случае *Router* определяет набор маршрутов при помощи *Switch*.

Каждый маршрут представляет собой объект *Route*, который имеет два основных атрибута: *path* (путь, который совпадает с адресом запроса) и *component* (компонент который отвечает за обработку запроса).

Так же стоит учитывать квалификаторы, *exact* (косвенное совпадение маршрута), *strict* (точное совпадение маршрута).

Дочерние маршруты. React поддерживает такого вида маршруты, которые могут являться подмаршрутами других маршрутов. Пример:

```
ReactDOM.render(  
  <Router>  
    <Switch>  
      <Route exact path="/" component={Home} />  
      <Route path="/about" component={About} />  
      <Route path="/products" component={Products} />  
      <Route component={NotFound} />  
    </Switch>  
  </Router>,  
  document.getElementById("app")  
)
```

К маршруту `products` например можно сопоставить вложенные маршруты, определив их в теле самого класса.

```

class Products extends React.Component{
  render(){
    return <div>
      <h2>Товары</h2>
      <Switch>
        <Route path="/products/phones" component={Phone} />
        <Route path="/products/tablets" component={Tablet} />
      </Switch>
    </div>;
  }
}

```

Получается что вложенные маршруты обрабатывают относительно своего главного маршрута, но при этом тело страницы будет модифицировано согласно компонента подмаршрута.

Создание ссылок. Навигацию по приложению можно осуществлять не только при помощи маршрутов, для этого так же существует поддержка ссылок:

```
const Link = ReactRouterDOM.Link;
```

Пример кода:

```

class Nav extends React.Component{
  render(){
    return <nav>
      <Link to="/">Главная</Link>
      <Link to="/about">О сайте</Link>
      <Link to="/products">Товары</Link>
    </nav>;
  }
}

ReactDOM.render(
  <Router>
    <div>
      <Nav />
      <Switch>
        <Route exact path="/" component={Home} />
        <Route path="/about" component={About} />
        <Route path="/products" component={Products} />
        <Route component={NotFound} />
      </Switch>
    </div>
  </Router>,
  document.getElementById("app")
)

```

Для определения блока навигации, по стандарту HTML5, добавляем наши ссылки в блок *nav*. Затем для каждой ссылки, с помощью атрибута *to*, определяем путь перехода.

После чего, компонент *Nav* помещаем в объект *Router*.

Существует объект `NavLink`. Который имеет дополнительно два атрибута, при помощи которых можно установить активный стиль ссылки: *activeClassName* и *activeStyle*.

```
<NavLink exact to="/" activeStyle={{color:"green",  
fontWeight:"bold"}}>Главная</NavLink>
```

Для стилизации ссылки через *activeStyle* допускается использование данных в формате `json`.

Параметры маршрутов. В добавок к `path` у маршрутов можно добавлять параметры. Пример:

```
<Route path="/products/:id" component={Product} />
```

для получения параметров, в компоненте необходимо обратиться к объекту:

```
const id= this.props.match.params.id;
```

параметр маршрута может быть необязательным:

```
<Route path="about/:id?" component={About} />
```

или вовсе ограниченным, при помощи регулярных выражений:

```
<Route path="/products/:id(\d+)" component={Product} />
```

Ссылки с параметрами. Если необходимо добавить к адресу ссылки параметр, в таком случае весь адрес помещается в фигурные скобки. Пример:

```
const phones = [{id: 1, name: "Apple iPhone 12 Pro"}];
```

```
...
```

```
<NavLink to={`/${products/${item.id}}`} ${item.name}</NavLink>
```

Парсинг строки запроса. Когда приложению приходит запрос, все его данные доступны в компоненте через объекты *location* и *match*, получаемые через объект *props*.

Основные свойства объекта *match*:

- *path*: шаблон `url` маршрута, выбранного для обработки запроса
- *url*: запрошенный путь с параметрами, без строки запроса
- *params*: набор параметров маршрута

Основные свойства объекта *location*:

- *pathname*: запрошенный путь `url` с параметрами, без строки запроса
- *search*: строка запроса

Для получения параметров маршрута применяется объект *this.props.match.params*, а для извлечения параметров строки запроса - объект *this.props.location.search*. Причем для извлечения параметров строки запроса необходимо использовать метод *URLSearchParams*. Пример:

```
<p>Match: {JSON.stringify(this.props.match)}</p>
<p>Location {JSON.stringify(this.props.location)}</p>
<p>Id: {this.props.match.params.id}</p>
<p>Name: {new URLSearchParams(this.props.location.search).get("name")}</p>
<p>Age: {new URLSearchParams(this.props.location.search).get("age")}</p>
```

Переадресация. В случае когда может потребоваться переадресация, необходимо использовать специальный объект *Redirect*. Для которого определены основных два параметра: *old* и *new*.

```
<Router>
  <div>
    <nav>
      <Link to="/">Home</Link>
      <Link to="/old">Old</Link> |
      <Link to="/new">New</Link>
    </nav>
    <Switch>
      <Route exact path="/" component={Home} />
      <Route path="/new" component={New} />
      <Redirect from="/old" to="/new" />
    </Switch>
  </div>
</Router>
```

В случае когда возникает необходимо в передаче параметров - определяется вспомогательный компонент, который получает, а затем передает их через редирект новому маршруту.

```
class Old extends React.Component{
  render(){
    return <Redirect to={`/new/${this.props.match.params.id}`} />;
  }
}
...
<Router>
  <div>
    <nav>
      <Link to="/old/123">Old</Link>
      <Link to="/new/456">New</Link>
    </nav>
    <Switch>
      <Route path="/new/:id" component={New} />
      <Route path="/old/:id" component={Old} />
    </Switch>
  </div>
</Router>
```

Хуки

Введение в хуки. Это функции которые позволяют определять и использовать состояние и другие возможности React без создания классов. Так же поддерживается создание своих хуков.

Стандартные хуки React:

- *useState*, предназначен для управления состоянием компонентов
- *useEffects*, предназначен для перехвата изменений компонентов, которые невозможно обработать внутри компонента
- *useContext*: позволяет подписываться на контекст в React
- *useReducer*, позволяет управлять лок. состоянием сложных компонентов
- *useCallback*, позволяет управлять фун-ми обратного вызова
- *useMemo*, пзволяет управлять кешированными значениями
- *useRef*, возвращает измененное значение (ссылка на DOM)
- *useImperativeHandle*, настраивает объект, который передается родительскому компоненту, при использовании ref
- *useLayoutEffect*, аналог. *useEffect* (вызывается синхронно, после всех изменений в структуре DOM)
- *useDebugValue*, предназначен для отображения некоторого значения в целях отладки

Как это можно использовать? Например у нас есть кнопка, и мы сделали класс который позволяет вести подсчет количества её нажатий. Для этого необходимо в конструкторе класса объявить переменную объекта state, определить метод-обработчик нажатия кнопки и метод который будет выполнять инкремент значения объекта state.

При использовании хука *useState* происходит все точно так же, однако с несколькими отличиями. Например обработка происходит не в классе компонента, а в функциональном компоненте. Поскольку React не позволяет реализацию хуков в классах, только в функциях. Во вторых, хук *useState* объявляет переменную(константу) типа массив, которая имеет вид:

```
const [count, setCount] = React.useState(0);
```


И здесь следует понимать что count - это переменная, а setCount - метод для её обработки. Идентификатор 0 в качестве фактического параметра, это значение по умолчанию для переменной.

При это так же определяется обработчик нажатия кнопки, однако без всяких this.

```
<script type="text/babel">
  function ClickButtonHook(props){
    const [count, setCount] = React.useState(0);
    const press = function(){
      setCount(count + props.increment);
    };
    return (<div>
      <button onClick={press}>Count</button>
      <div>Counter: {count}<br /> Increment: {props.increment}</div>
    </div>);
  }

  ReactDOM.render(
    <ClickButtonHook increment={2} />,
    document.getElementById("app")
  )
</script>
```

Стрелочный вид:

```
const ClickButtonHook = (props)=>{
  const [count, setCount] = React.useState(0);

  const press= () => setCount(count + props.increment);
  return <div>
    <button onClick={press}>Count</button>
    <div>Counter: {count}<br /> Increment: {props.increment}</div>
  </div>;
}
```

Ограничения хуков. Хуки вызываются только на верхнем уровне компонента. Их нельзя вызвать в теле циклов, условных конструкций, внутри стандартных функций. Они вызываются ТОЛЬКО из функциональных компонентов React, либо из других хуков.

2 Постановка задачи

Задача: создание пошаговой формы при помощи React.

Цель: научиться пользоваться базовыми инструментами React.

По сколько пошаговые формы позволяют разбить ввод данных на отдельные этапы. Рассмотрим в этом случае, как сделать подобную форму.

Допустим, пользователь вводит сначала имя, потом email-адрес и в конце пароль. И при этом пользователю последовательно отображаются форма ввода имени, затем форма ввода email и форма ввода пароля.

3 Выполнение задания

Ссылка на проект: <https://github.com/dEROZA/cw-programming-react>

Структура программы показана на рисунке 1. В файле index.html содержится код скелета (рисунок 2) необходимый для пользовательского интерфейса, согласно поставленной задаче. Описание файлов проекта:

- *headrt.jsx*, предназначен для формирования заглавного блока
- *body.jsx*, блок для тела страницы, для форм и приема данных
- *footer.jsx*, нижний блок, который выводит текущее время
- *Каталог external*, который хранит библиотеки и стили react, bootstrap
- *PHP скрипт*, необходим для приема ajax запросов формата Json

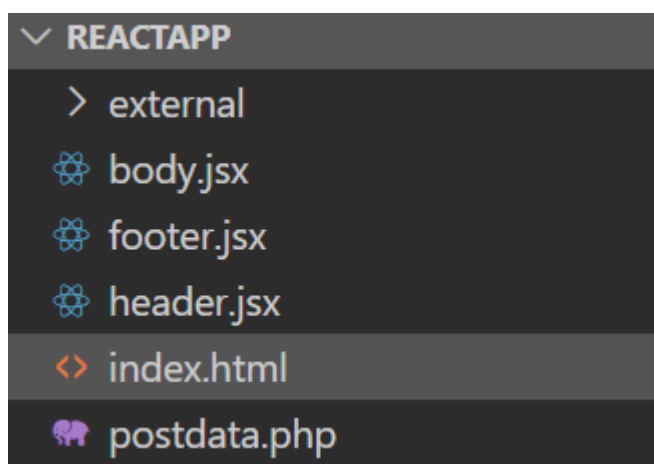


Рисунок 1 - Структура программы

Описание ключевых особенностей кода. При вводе данных в форму, функция `handleChange` проверяет эти данные хранящиеся в элементе `input` при помощи регулярных выражений:

```
handleChange = event => {
  const {name, value} = event.target
  this.setState({
    [name]: value
  })
  let st = this.state.currentStep;
  switch (st) {
    case 1: {
      this.state.activestate = value.match(/^(\\w+\\s?\\w+)$/ ) !== null;
      break;
    } //console.log(name, value, value.match(/^(\\w+\\s?\\w+)$/)); break;
    case 2: {
      this.state.activestate = value.match(/^(\\w+\\@\\w+\\.\\w+)$/ ) !== null;
      break;
    } //console.log(name, value, value.match(/^(\\w+\\@\\w+\\.\\w+)$/)); break;
    case 3: {
      this.state.activestate = value.match(/^(\\w+|\\~+|\\*+|\\@+|\\#+)$/ ) !== null;
      break;
    } //console.log(name, value, value.match(/^(\\w+|\\~+|\\*+|\\@+|\\#+)$/)); break;
    default: break;
  }
}
```

Свойство *activestate*, объекта *state*, является условием для переключения режима атрибута *disabled* для объектов *button* (кнопки назад, вперед, отправить). Благодаря этому пользователь может быть своевременно информирован о том что данные которые он вводит в форму могут быть неверными или не соответствовать шаблону ввода для конкретного формата.

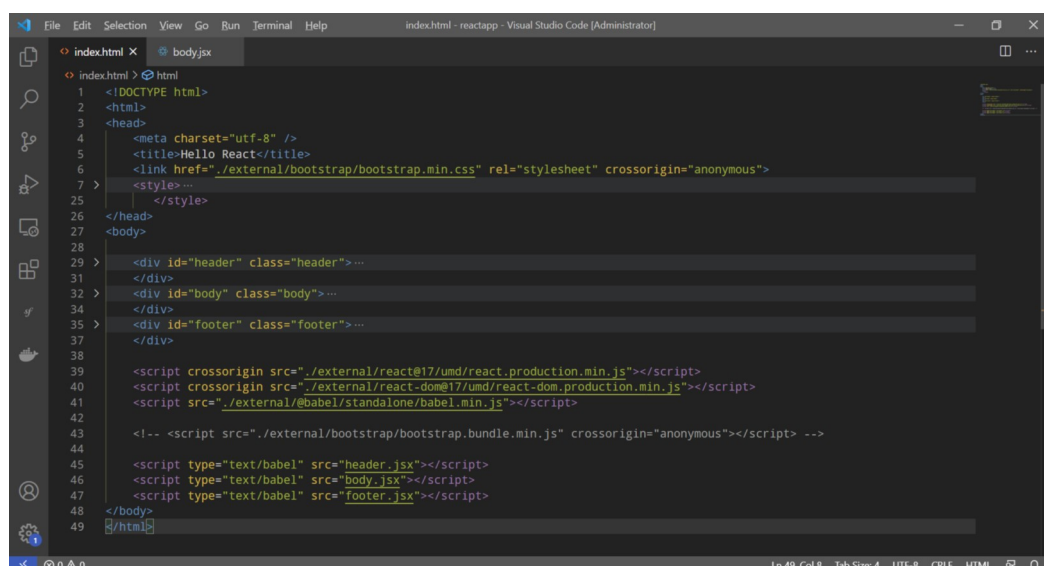


Рисунок 2 - Тело файла *index.html*

Блок head

```
<head>
  <meta charset="utf-8" />
  <title>Hello React</title>
  <link href="./external/bootstrap/bootstrap.min.css"
rel="stylesheet" crossorigin="anonymous">
  <style>
    .header {
      position: relative;
      background: #ffebe9; /* Цвет фона */
      height: 75px; /* Высота блока */
    }
    .footer {
      position: relative;
      background: #f2ffe8; /* Цвет фона */
      height: 75px; /* Высота блока */
    }
    .body {
      position: relative;
      background: hsl(266, 100%, 95%); /* Цвет фона */
      height: 200px; /* Высота блока */
      border-top: 1px solid;
      border-bottom: 1px solid;
    }
  </style>
</head>
```

Данный блок в основном содержит информацию, которая необходима в разметке для построения визуальной части.

Блок body

```
<div id="header" class="header">
  <div id="announcement"></div>
</div>
<div id="body" class="body">
  <div id="register"></div>
</div>
<div id="footer" class="footer">
  <div id="timer"></div>
</div>
```

За построение тела для форм регистрации, а так же для всех остальных вспомогательных блоков (header, footer) отвечает этот блок.

Так же в нем располагаются HTML-теги для подключения необходимых библиотек: React и Babel:

```
<script type="text/babel" src="header.jsx"></script>
<script type="text/babel" src="body.jsx"></script>
<script type="text/babel" src="footer.jsx"></script>
```

Код файла header.jsx

```
const styleAnnouncementBlock = {
  fontFamily: 'Consolas',
  position: 'absolute',
  top: '15px',
  left: '25px'
};

class Announcement extends React.Component {
  render() {
    return (
      <h1 style={styleAnnouncementBlock}>Привет, Гость!</h1>
    );
  }
}

ReactDOM.render(
  <Announcement />,
  document.getElementById("announcement")
)
```

На данном этапе определяется конструкция класса для стиля CSS. Согласно нотации JSX. Затем определяет класс компонента. После чего вызов метода *ReactDOM.render* производит рендеринг React-элемента.

Код файла footer.jsx

```
const styleTimerBlock = {
  fontFamily: 'Consolas',
  position: 'absolute',
  bottom: '15px',
  right: '15px',
  paddingRight: '25px'
};

function tick() {
  ReactDOM.render(
    <div style={styleTimerBlock}>
      <h2>Текущее время {new Date().toLocaleTimeString()}</h2>
    </div>,
    document.getElementById("timer")
  );
}

tick(); setInterval(tick, 1000);
```

В этом файле хранится метод запуска таймера, и функциональный компонент, который обновляется в заданном таймером интервале.

Код файла body.jsx

В данном файле хранятся основные React компоненты, которые отвечают за форму регистрации пользователя. Основным компонентом является *UserForm*. Данный компонент отвечает за общее состояние формы регистрации, методы обработчиков событий нажатия кнопок(*назад*, *вперед*, *отправить*), служебные методы валидации вводимых данных, рендер методы и пр.

Остальные компоненты(*UserNameStep*, *EmailStep*, *PasswordStep*) являются дежурными, метод *render()* которых вызывается согласно объекта состояния, шага на котором находится пользователь.

Порядок вызова компонентов:

- *UserForm*
- *UserNameStep*
- *EmailStep*
- *PasswordStep*

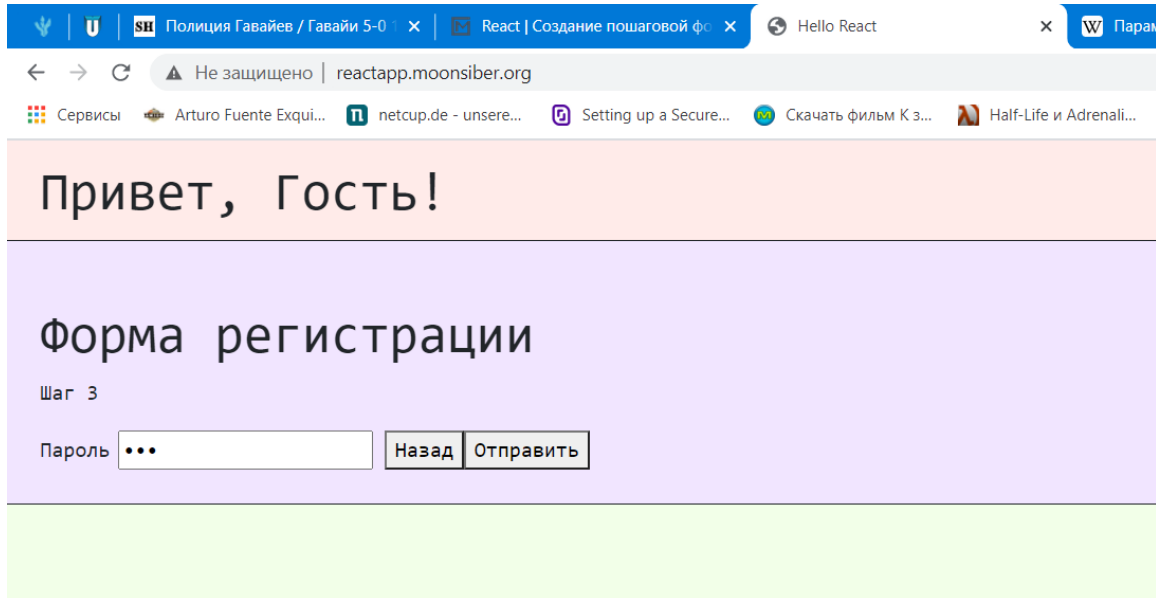
В случае нормального завершения процесса регистрации, когда все данные введены пользователем корректно и проверены обработчиком JS скрипта - посредством технологии Ajax, данные отправляются на удаленный сервер.

Код запроса React к удаленному ресурсу(метод POST):

```
const requestOptions = {
  method: 'POST',
  headers: {
    'Content-Type': 'application/json',
    'Authorization': 'Bearer eyJhbGciOiJIUzUxMiIsInR5cCI6IkpXVCJ9'
  },
  body: JSON.stringify({
    uname: this.state.username,
    email: this.state.email,
    upass: this.state.password
  })
};
fetch('http://reactapp.moonsiber.org/postdata.php', requestOptions)
  .then(response => console.log(response));
```

Запрос сформирован с учетом спецификации REST архитектуры, по принципам которой функционирует внешняя API система.

Успешный ввод данных показан на рисунке 3. После нажатия кнопки отправить, данные передадутся на микросервис API. Система API учитывая заголовок токена, принимает данные, повторно проводит валидацию на корректность и затем помещает их в БД (рисунок 4).



Привет, Гость!

Форма регистрации

Шаг 3

Пароль

Рисунок 3 - Шаг 3й, отправка данных на внешний API

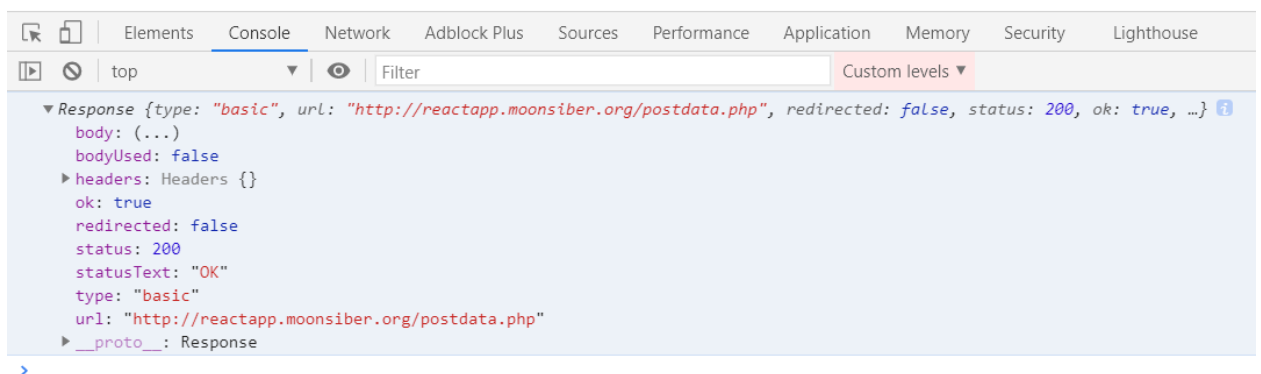


Рисунок 4 - Результат успешной отправки запроса к API

Заключение

В общем случае React является отличной библиотекой для построения пользовательских интерфейсов и механик. Поддержка ООП парадигмы в старом добром JavaScript - это взгляд по истине свежий.

Конечно, любой инструмент который упрощает разработку разделяя зоны ответственности будет довольно эффективным. React является таковым.

Однако стоит учитывать и негативные факторы, к которым можно отнести увеличение сложности отладки.

В конечном итоге, React на сегодня набирает популярность среди людей которые понимают JavaScript и умеют его использовать. Именно это надо уметь и знать, если производитель SPA был уверен в том что его клиенты видят свой продукт отзывчивым, простым в использовании и экономящие ресурсы их устройств, не предъявляя при этом высоких требований к производительности.

Список использованных источников

1. Сайт о программировании. [Электронный ресурс] : Руководство по React. <https://metanit.com/web/react/> (дата обращения 31.03.2021).
2. Официальная документация по React. [Электронный ресурс] : Основные понятия React. <https://ru.reactjs.org/docs/getting-started.html> (дата обращения 31.03.2021).