GeekBrains

Django REST Framework

Работа с формами. Сборка проекта для production

На этом уроке

1. Узнаем, какие варианты сборки для production существуют.
2. Научимся собирать frontend вместе с backend.
3. Узнаем, как работать с формами в React.
4. Научимся делать POST- и DELETE-запросы на сервер.

Оглавление

Работа с формами

Введение

В курсе мы подробно рассмотрели работу с backend-частью приложения. Также мы изучили принцип one-way data flow для создания простого приложения на React. На этом занятии мы рассмотрим развитие принципа one-way data flow, как происходит взаимодействие между компонентами и как создаются компоненты для форм на React.

В качестве демонстрационного примера мы добавим в проект library, во frontend-часть на React, возможность создавать новые книги, а также удалять их. Это позволит:

- поработать с формами на React;

- поработать со связанными компонентами (списком авторов и тегом `select`);
- поработать с POST- и DELETE-запросами на сервер.

Внимание! *В коде frontend-части приложения мы не будем использовать дополнительные библиотеки для работы с формами, чтобы лучше освоить базовые умения. Наш код будет содержать некоторое дублирование.*

Удаление книг. Отправка DELETE-запроса на сервер

Добавим возможность удалять книги на стороне клиента. Для этого рядом с каждой книгой выведем кнопку Delete. Внесём следующие изменения в файл Book.js:

```
import React from 'react'


const BookItem = ({item}) => {
    return (
        <tr>
            <td>{item.id}</td>
            <td>{item.name}</td>
            <td>{item.author.name}</td>
            <td><button type='button'>Delete</button></td>
        </tr>
    )
}


const BookList = ({items}) => {
    return (
        <table>
            <tr>
                <th>ID</th>
                <th>NAME</th>
                <th>AUHTOR</th>
                <th></th>
            </tr>
            {items.map((item) => <BookItem item={item} />)}
        </table>
    )
}


export default BookList
```

/frontend/src/components/Book.js

В `BookList` мы добавили пустой тег `th` для заголовка, а в `BookItem` — кнопку `button` с надписью `Delete`.

Данные о книгах по принципу one-way data flow находятся в классе `App`, но кнопка «Удалить» находится в компоненте `BookItem`. В этом случае мы создаём метод для работы с данными в классе `App` — там же, где находятся данные, с которыми мы работаем. Затем передаём этот метод как callback до компонента нижнего уровня. Сделаем это по шагам. Сначала в класс `App` добавим метод для удаления книги:

```
class App extends React.Component {
...

  deleteBook(id) {
    const headers = this.get_headers()
    axios.delete(`http://127.0.0.1:8000/api/books/${id}`, {headers, headers})
        .then(response => {
          this.setState({books: this.state.books.filter((item)=>item.id !==
id)})
        }).catch(error => console.log(error))
  }
...
```

/frontend/src/App.js

Сначала мы получаем заголовки для авторизации и отправляем DELETE-запрос:

```
const headers = this.get_headers()
    axios.delete(`http://127.0.0.1:8000/api/books/${id}`, {headers, headers})
```

При удалении объекта на сервере мы удаляем объект на клиенте и обновляем состояние компонента `App`:

```
this.setState({books: this.state.books.filter((item)=>item.id !== id)})
```

С помощью метода `filter` мы выбираем все книги, кроме удалённой, и обновляем состояние нашего приложения.

После создания функции передаём её до компонента `BookItem`. Сначала в App.js передаём в компонент `BookList`:

```
class App extends React.Component {
...
  render() {
    return (
```

```
        ...
              <Route exact path='/books' component={() => <BookList
items={this.state.books} deleteBook={(id)=>this.deleteBook(id)} />} />
...
```

/frontend/src/App.js

Затем в файле Book.js вносим следующие изменения:

```
import React from 'react'


const BookItem = ({item, deleteBook}) => {
    return (
        <tr>
            <td>{item.id}</td>
            <td>{item.name}</td>
            <td>{item.author.name}</td>
            <td><button onClick={()=>deleteBook(item.id)}
type='button'>Delete</button></td>
        </tr>
    )
}


const BookList = ({items, deleteBook}) => {
    return (
        <table>
            <tr>
                <th>ID</th>
                <th>NAME</th>
                <th>AUHTOR</th>
                <th></th>
            </tr>
            {items.map((item) => <BookItem item={item} deleteBook={deleteBook}
/>)}
        </table>
    )
}


export default BookList
```

/frontend/components/Book.js

В `BookList` мы принимаем `deleteBook` и передаём её в `BookItem`. Затем в `BookItem` мы принимаем `deleteBook`. А дальше связываем эту функцию с нажатием на кнопку:

```
<button onClick={()=>deleteBook(item.id)} type='button'>Delete</button>
```

Для каждой книги мы передаём свой id. Теперь при нажатии на кнопку рядом с каждой книгой будет срабатывать callback, написанный в классе `App`.

Создание книги. Отправка POST-запроса на сервер

Теперь мы знаем, где создавать функцию для работы с данными и как передавать её в компоненты нижнего уровня по принципу one-way data flow. Для создания книги нам ещё понадобится компонент формы для ввода данных и сохранения введённых значений.

Создание формы

Работа с формами описана в официальной документации React. Применим это для нашей задачи. В папке components создадим новый файл BookForm.js со следующим содержанием:

```
import React from 'react'


class BookForm extends React.Component {
    constructor(props) {
      super(props)
      this.state = {name: '', author: 0}
    }

    handleChange(event)
    {
        this.setState(
                {
                    [event.target.name]: event.target.value
                }
            );
    }

    handleSubmit(event) {
      console.log(this.state.name)
      console.log(this.state.author)
      event.preventDefault()
    }

    render() {
      return (
        <form onSubmit={(event)=> this.handleSubmit(event)}>
            <div className="form-group">
            <label for="login">name</label>
                <input type="text" className="form-control" name="name"
value={this.state.name} onChange={(event)=>this.handleChange(event)} />
            </div>

        <div className="form-group">
```

```
            <label for="author">author</label>

            <input type="number" className="form-control" name="author"
value={this.state.author} onChange={(event)=>this.handleChange(event)} />


          </div>
          <input type="submit" className="btn btn-primary" value="Save" />
        </form>
      );
    }
  }

  export default BookForm
```

/frontend/components/BookForm.js

Рассмотрим этот код по частям:

```
class BookForm extends React.Component {
    constructor(props) {
      super(props)
      this.state = {name: '', author: 0}
    }
```

Чтобы запомнить введённые данные, мы создали компонент `React` на основе класса. В конструкторе мы определили состояние `name` — название книги, `author` — id автора.

Внимание! *Для последовательного рассмотрения темы мы сначала будем вводить id автора, а затем после проверки сохранения сделаем выпадающий список с именами авторов.*

```
handleChange(event)
    {
        this.setState(
                {
                    [event.target.name]: event.target.value
                }
            );
    }
...
<input type="text" className="form-control" name="name" value={this.state.name}
onChange={(event)=>this.handleChange(event)} />
            </div>

        <div className="form-group">
            <label for="author">author</label>

            <input type="number" className="form-control" name="author"
```

```
value={this.state.author} onChange={(event)=>this.handleChange(event)} />
```

Метод `handleChange` будет срабатывать при каждом изменении значений в полях для ввода формы. Мы будем сохранять изменённые данные в состоянии формы. Ключом будет имя `input`'а (`event.target.name`), а значением — `event.target.value`.

```
handleSubmit(event) {
      console.log(this.state.name)
      console.log(this.state.author)
      event.preventDefault()
    }
```

Метод `handleSubmit` будет выполняться после отправки формы. Для начала проверим, что мы запоминаем название книги и автора в состоянии формы. `event.preventDefault()` отменяет отправку формы, так как мы сами обрабатываем отправку на стороне клиента.

Форма готова. Теперь в файле Book.js в компоненте `BookList` добавим ссылку для перехода на страницу с формой:

```
import React from 'react'
import {Link} from 'react-router-dom'

...

const BookList = ({items, deleteBook}) => {
    return (
        <div>
        <table>
            <tr>
                <th>ID</th>
                <th>NAME</th>
                <th>AUHTOR</th>
                <th></th>
            </tr>
            {items.map((item) => <BookItem item={item} deleteBook={deleteBook}
/>)}
        </table>
        <Link to='/books/create'>Create</Link>
        </div>
    )
}


export default BookList
```

/frontend/src/components/Book.js

Для маршрутизации на стороне клиента мы используем тег `Link`:

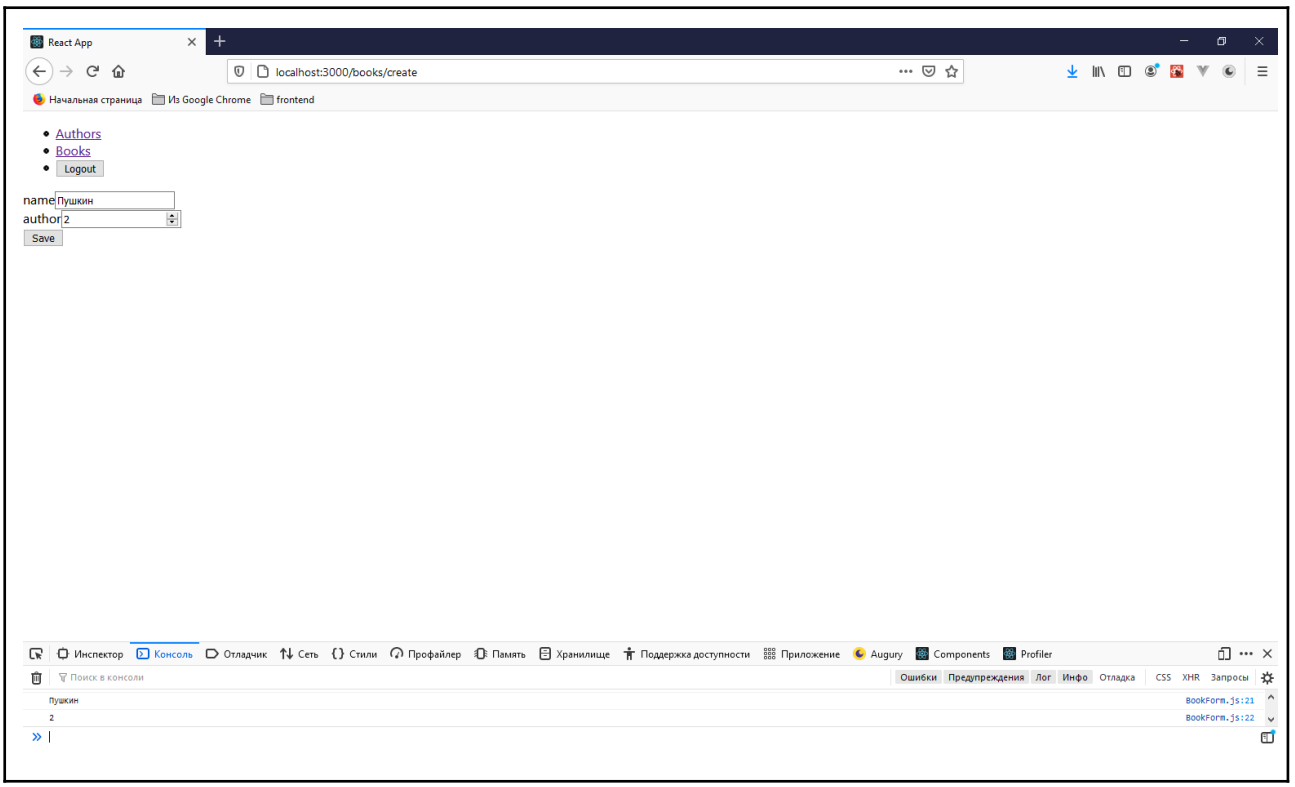
```
<Link to='/books/create'>Create</Link>
```

Теперь в класс `App` добавляем маршрут для связи адреса с компонентом формы:

```
...
import {BrowserRouter, Route, Switch, Redirect, Link} from 'react-router-dom'
...

class App extends React.Component {
...

  render() {
    return (
        <div className="App">
          ...
              <Route exact path='/books/create' component={() => <BookForm />}
/>
              <Route exact path='/books' component={() => <BookList
items={this.state.books} deleteBook={(id)=>this.deleteBook(id)} />} />
            ...
    )
  }
}

export default App
```

/frontend/src/App.js

Теперь по нажатию на кнопку create на странице со списком книг мы переходим на страницу с формой. Если всё верно, то введённые данные после отправки формы выводятся в консоль:



Callback для отправки POST-запроса

В классе `App` создадим функцию для создания новой книги:

```
class App extends React.Component {

 ...

  createBook(name, author) {
    const headers = this.get_headers()
    const data = {name: name, author: author}
    axios.post(`http://127.0.0.1:8000/api/books/`, data, {headers, headers})
        .then(response => {
          let new_book = response.data
          const author = this.state.authors.filter((item) => item.id ===
new_book.author)[0]
          new_book.author = author
          this.setState({books: [...this.state.books, new_book]})
        }).catch(error => console.log(error))
  }
...
```

/frontend/src/App.js

Рассмотрим этот код подробнее:

```
const headers = this.get_headers()
const data = {name: name, author: author}
axios.post(`http://127.0.0.1:8000/api/books/`, data, {headers, headers})
```

Сначала мы формируем заголовки и данные запроса и отправляем POST-запрос на сервер:

```
let new_book = response.data
const author = this.state.authors.filter((item) => item.id ===
new_book.author)[0]
new_book.author = author
```

После успешного выполнения запроса сервер в ответе вернёт созданный объект книги. В этом объекте поле `author` будет содержать id автора. Поэтому, прежде чем добавлять эти данные в состояние клиента, мы ищем объект автора по его id и заменяем id в объекте на найденный объект автора.

```
this.setState({books: [...this.state.books, new_book]})
```

После этого мы обновляем состояние нашего приложения. С помощью `...this.state.books` мы распаковываем имеющийся список книг и добавляем к нему сформированный объект `new_book`.

Теперь осталось передать callback в компонент формы:

```
class App extends React.Component {
...

  render() {
    return (
        ...
              <Route exact path='/books/create' component={() => <BookForm
createBook={(name, author) => this.createBook(name, author)} />}  />
...
    )
  }
}
```

/frontend/src/App.js

Далее заменим вывод в консоль на вызов callback:

```
class BookForm extends React.Component {
    ...
```

```
    handleSubmit(event) {
      this.props.createBook(this.state.name, this.state.author)
      event.preventDefault()
    }
...
```

/frontend/src/components/BookForm.js

Вывод авторов в выпадающем списке

Теперь мы можем удалять и создавать книги, но для создания нужно знать id автора. Это неудобно. Изменим форму для использования тега `select`, в котором будут выводиться имена авторов и сохраняться id выбранного автора.

В классе `App` у нас есть список авторов. Передадим его в компонент `BookForm`:

```
...
<Route exact path='/books/create' component={() => <BookForm
authors={this.state.authors} createBook={(name, author) => this.createBook(name,
author)} />}  />
...
```

/frontend/src/App.js

Далее изменим код в BookForm.js следующим образом:

```
import React from 'react'


class BookForm extends React.Component {
    constructor(props) {
      super(props)
      this.state = {name: '', author: props.authors[0].id}
    }

    handleChange(event)
    {
        this.setState(
                {
                    [event.target.name]: event.target.value
                }
            );
    }

    handleSubmit(event) {
      this.props.createBook(this.state.name, this.state.author)
      event.preventDefault()
    }
```

```
    render() {
      return (
        <form onSubmit={(event)=> this.handleSubmit(event)}>
            <div className="form-group">
            <label for="login">name</label>
                <input type="text" className="form-control" name="name"
value={this.state.name} onChange={(event)=>this.handleChange(event)} />
            </div>

        <div className="form-group">
            <label for="author">author</label>

            <select name="author" className='form-control'
onChange={(event)=>this.handleChange(event)}>
                {this.props.authors.map((item)=><option
value={item.id}>{item.name}</option>)}
            </select>

          </div>
          <input type="submit" className="btn btn-primary" value="Save" />
        </form>
      );
    }
  }

  export default BookForm
```

/frontend/src/components/BookForm.js

Рассмотрим внесённые изменения:

```
constructor(props) {
      super(props)
      this.state = {name: '', author: props.authors[0].id}
    }
```

В конструкторе для автора мы задаём первый элемент списка. Он первым загрузится в `select` и, если значение не изменится, то будет выбран именно он.

```
render() {
      return (
        ...

        <div className="form-group">
            <label for="author">author</label>

            <select name="author" className='form-control'
```

```
onChange={(event)=>this.handleChange(event)}>
                {this.props.authors.map((item)=><option
value={item.id}>{item.name}</option>)}
            </select>
...
```

При отрисовке формы мы создаём тег `select`, элементы которого (теги `option`) мы формируем на основе переданного списка авторов.

Теперь мы можем выбирать автора из выпадающего списка и более удобно создавать новую книгу.

Сборка проекта для production

На следующем занятии мы на практике рассмотрим процесс развёртывания нашего проекта с помощью Docker и Docker Compose. Для начала разберёмся, как frontend-часть сайта собирается вместе с backend-частью.

Есть два основных варианта развёртывания проекта. Для обоих нужно подготовить (собрать) frontend-часть. Первый вариант заключается в сборке frontend'а вместе с backend'ом и использовании одного балансировщика нагрузки. Второй вариант: использовать frontend отдельно от backend'а и применить два балансировщика.

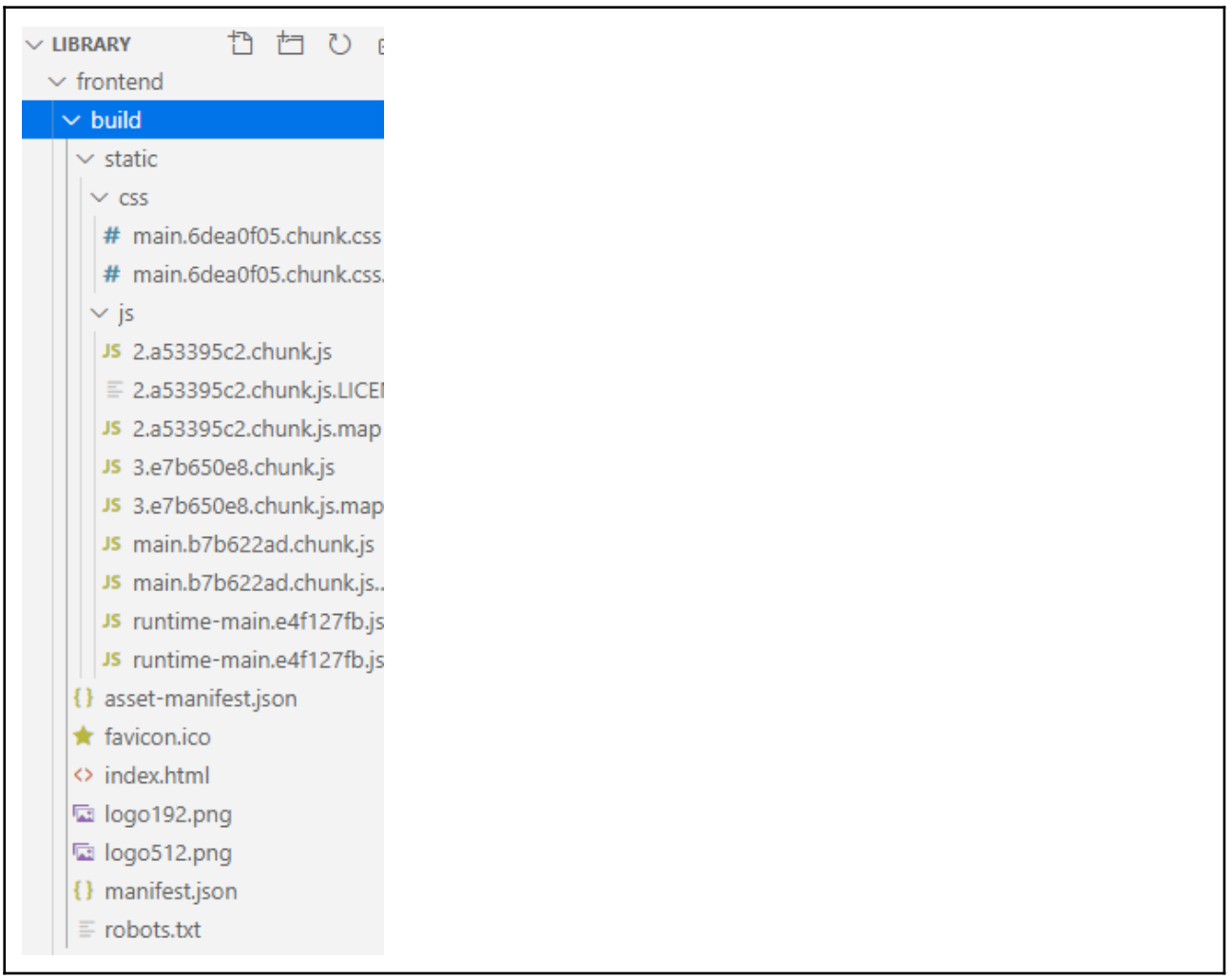
Сборка frontend-части

Какой бы способ развёртывания мы ни выбрали, предварительно мы должны сделать production build для frontend-части приложения. Для его создания мы в папке frontend запускаем команду:

```
npm run build
```

Процесс сборки происходит следующим образом: начиная с файла /frontend/src/index.js, сборщик будет определять зависимые файлы стилей, JavaScript, шрифтов, HTML-страниц и так далее. Когда зависимости определены, сборщик упакует всё удобным способом и создаст структуру собранного проекта.

После сборки в папке /frontend/build будет создана следующая структура файлов и папок:



Наше приложение содержит файл index.html, к которому подключены все нужные файлы стилей, кода на JavaScript, изображений и так далее.

Сборка frontend вместе с backend

После окончания процесса сборки в Django-проекте мы можем создать `TemplateView` с указанием пути до получившегося файла index.html и настроить файлы статики. Таким образом, при переходе на `TemplateView` Django загрузит одну страницу, а все последующие взаимодействия будут происходить на стороне React.

Сборка frontend отдельно от backend

При отдельной сборке мы можем настроить один балансировщик нагрузки (например, nginx) на отрисовку статической страницы index.html, которую собрал сборщик проекта. А второй балансировщик настроить на работу с Django-проектом. Таким образом, у нас будут два независимых сервера, при этом сервер с frontend будет перенаправлять запросы к серверу backend.

Итоги

На этом занятии мы рассмотрели работу с формами React, повторили принцип one-way data flow и узнали варианты сборки проекта для production. На следующем занятии мы на практике изучим процесс развёртывания проекта с помощью Docker и Docker Compose.

Глоссарий

Nginx — это HTTP-сервер и обратный прокси-сервер, почтовый прокси-сервер, а также TCP/UDP прокси-сервер общего назначения, изначально написанный Игорем Сысоевым.

Callback — функция обратного вызова

Дополнительные материалы

1. React Forms.
2. One way data flow React.

Практическое задание

Работа с формами на React. В этой самостоятельной работе мы тренируем умения:

- работать с формами на React;
- отправлять POST- и DELETE-запросы.

Смысл: работать со всеми типами запросов в своих проектах.

Последовательность действий

1. В проекте добавить возможность создавать и удалять проекты.
2. Добавить возможность создавать и удалять ToDo.
3. Добавить поиск по части названия проекта.
4. Все запросы на сервер рекомендуется делать в главном приложении.
5. Для взаимодействия с главным приложением передавать callback.
6. * Добавить возможность изменять проекты.