GeekBrains

Django REST Framework

Авторизация на стороне клиента

На этом уроке

1. Узнаем, как настроить авторизацию на стороне клиента.
2. Узнаем, где хранить токен авторизации.
3. Научимся прикладывать токен к запросу.
4. Научимся сохранять токен в localStorage и cookies.

Оглавление

Получение токена авторизации с backend

Введение

На этом занятии мы реализуем на React авторизацию на стороне клиента для демонстрационного проекта Library. Для других проектов этот процесс будет похожим. Он включает следующие шаги:

1. Создать форму для авторизации на React. В неё пользователь будет вводить данные авторизации (логин и пароль).
2. Получить логин и пароль пользователя после отправки формы и отправить с ними запрос на сервер для получения токена авторизации данного пользователя.

3. Получить токен авторизации в ответе с сервера.
4. Сохранить токен авторизации в localStorage или в cookies.
5. Прикладывать токен авторизации ко всем запросам на сервер.

Это позволит:

- идентифицировать пользователя на стороне backend по токену;
- хранить токен на стороне клиента для новых взаимодействий с приложением, не авторизовываться каждый раз после закрытия браузера и вкладки.

Рассмотрим весь процесс по шагам на нашем демонстрационном проекте.

Подготовка проекта

На предыдущем занятии мы создали backend-часть нашего проекта, а на занятии по Routing SPA — frontend-часть с маршрутизацией. Соединим их.

Установим библиотеку Axios для отправки запросов:

```
npm install axios
```

Изменим код в файле App.js следующим образом:

```
import React from 'react'
import AuthorList from './components/Author.js'
import BookList from './components/Book.js'
import AuthorBookList from './components/AuthorBook.js'
import {BrowserRouter, Route, Switch, Redirect, Link} from 'react-router-dom'
import axios from 'axios'


const NotFound404 = ({ location }) => {
  return (
    <div>
        <h1>Страница по адресу '{location.pathname}' не найдена</h1>
    </div>
  )
}


class App extends React.Component {

  constructor(props) {
    super(props)
    this.state = {
      'authors': [],
```

```
      'books': []
    }
  }

  load_data() {
    axios.get('http://127.0.0.1:8000/api/authors/')
        .then(response => {
            this.setState({authors: response.data})
        }).catch(error => console.log(error))

    axios.get('http://127.0.0.1:8000/api/books/')
        .then(response => {
            this.setState({books: response.data})
        }).catch(error => console.log(error))
  }

  componentDidMount() {
    this.load_data()
  }

  render() {
    return (
        <div className="App">
          <BrowserRouter>
             <nav>
            <ul>
            <li>
            <Link to='/'>Authors</Link>
            </li>
            <li>
            <Link to='/books'>Books</Link>
            </li>
            </ul>
            </nav>
            <Switch>
                          <Route exact path='/' component={() => <AuthorList
items={this.state.authors} />}  />
                        <Route exact path='/books' component={() => <BookList
items={this.state.books} />} />
              <Route path="/author/:id">
                <AuthorBookList items={this.state.books} />
              </Route>
              <Redirect from='/authors' to='/' />
              <Route component={NotFound404} />
            </Switch>
          </BrowserRouter>
        </div>
    )
  }
}

export default App;
```

/library/frontend/src/App.js

Мы добавили в класс `App` метод `load_data`, который получает данные с сервера. И вызвали этот метод в методе `componentDidMount`.

Далее на стороне backend'а нам нужно установить `django-cors-headers` для разрешения запросов с frontend'а:

```
pip install django-cors-headers
```

После этого в settings.py добавляем пакет в `INSTALLED_APPS`:

```
INSTALLED_APPS = [
    ...
    'corsheaders',
    ...
]
```

/library/settings.py

Добавляем `middleware`:

```
MIDDLEWARE = [
    ...
    'corsheaders.middleware.CorsMiddleware',
    'django.middleware.common.CommonMiddleware',
    ...
]
```

/library/settings.py

И указываем разрешённые хосты:

```
CORS_ALLOWED_ORIGINS = [
    "http://localhost:3000",
]
```

/library/settings.py

Немного изменим код `Serializer` для модели `book`, чтобы в поле автора выводился объект целиком, а не его идентификатор. Код в файле serializers.py будет иметь следующий вид:

```
from rest_framework import serializers
from .models import Author, Book

class AuthorSerializer(serializers.ModelSerializer):
    class Meta:
        model = Author
        fields = '__all__'


class BookSerializer(serializers.ModelSerializer):
    author = AuthorSerializer()
    class Meta:
        model = Book
        fields = '__all__'
```

/mainapp/serializers.py

Далее запускаем backend-сервер:

```
python manage.py runserver
```

И тестовый сервер frontend:

```
npm run start
```

Теперь сайт работает, данные тянутся с backend и всё готово для дальнейшей работы.

Создание формы авторизации

Внимание! *Подробнее работу с формами на React мы рассмотрим на следующих занятиях. А на этом занятии мы будем считать, что пользователь уже создан, и не будем работать с формой регистрации пользователя.*

В папке components создадим файл Auth.js со следующим кодом:

```
import React from 'react'


class LoginForm extends React.Component {
    constructor(props) {
      super(props)
      this.state = {login: '', password: ''}
    }
```

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

    handleSubmit(event) {
      console.log(this.state.login + ' ' + this.state.password)
      event.preventDefault()
    }

    render() {
      return (
        <form onSubmit={(event)=> this.handleSubmit(event)}>
                        <input  type="text"  name="login"  placeholder="login"
value={this.state.login} onChange={(event)=>this.handleChange(event)} />
                  <input type="password" name="password" placeholder="password"
value={this.state.password} onChange={(event)=>this.handleChange(event)} />
            <input type="submit" value="Login" />
        </form>
      );
    }
  }

  export default LoginForm
```

/frontend/src/components/Auth.js

Мы создали React-компонент для работы с формой. Рассмотрим этот код подробнее:

```
constructor(props) {
      super(props)
      this.state = {login: '', password: ''}
    }
```

Так как в форму пользователь будет вводить данные, мы будем использовать компонент с состоянием. В конструкторе мы создали состояние, в котором будут храниться login и password:

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

```
    }
```

Метод `handleChange` принимает в себя `event` — это событие, которое произойдёт при вводе данных в поля формы. Этот метод будет менять состояние `event.target.name` и записывать в него `event.target.value`. Так как `event.target` — это `input`, в который будут вводиться данные, его `name` будет либо `login`, либо `password`, а `value` — соответствующее введённое значение. Далее мы свяжем этот метод с нашими input'ами для ввода данных:

```
handleSubmit(event) {
      console.log(this.state.login + ' ' + this.state.password)
      event.preventDefault()
    }
```

Метод `handleSubmit` будет выполняться при отправке формы. В нём мы проверим, что правильно получили `login` и `password`, которые ввёл пользователь. `event.preventDefault()` отменит отправку формы. Это нужно, так как мы сами будем отправлять запрос на сервер с помощью Axios.

```
render() {
      return (
        <form onSubmit={(event)=> this.handleSubmit(event)}>
            <input type="text" name="login" placeholder="login"
value={this.state.login} onChange={(event)=>this.handleChange(event)} />
            <input type="password" name="password" placeholder="password"
value={this.state.password} onChange={(event)=>this.handleChange(event)} />
            <input type="submit" value="Login" />
        </form>
      );
    }
```

Метод `render` отрисовывает компонент формы. Событие `onSubmit` формы мы связываем с методом `handleSubmit`, а событие `onChange` на `input-ax` — с методом `handleChange`. Таким образом, при вводе данных будут меняться `login` и `password` в `state` компонента, а после отправки формы мы будем использовать введённые значения.

Теперь в App.js подключим компонент `LoginForm` и создадим для него новую страницу:

```
import LoginForm from './components/Auth.js'
...
render() {
    return (
        <div className="App">
          <BrowserRouter>
            <nav>
             <ul>
```

```
            <li>
            <Link to='/'>Authors</Link>
            </li>
            <li>
            <Link to='/books'>Books</Link>
            </li>
              <li>
              <Link to='/login'>Login</Link>
              </li>
            </ul>
           </nav>
            <Switch>
                          <Route exact path='/' component={() => <AuthorList
items={this.state.authors} />} />
                        <Route exact path='/books' component={() => <BookList
items={this.state.books} />} />
              <Route exact path='/login' component={() => <LoginForm />} />
              <Route path="/author/:id">
                <AuthorBookList items={this.state.books} />
              </Route>
              <Redirect from='/authors' to='/' />
              <Route component={NotFound404} />
            </Switch>
          </BrowserRouter>
        </div>
    )
  }
...
```

/frontend/src/App.js

Теперь мы можем переходить на страницу с авторизацией, а при вводе данных и нажатии кнопки Login в консоль выводятся логин и пароль, которые ввёл пользователь.

Получение токена авторизации

В App.js в классе `App` создадим метод для получения токена авторизации и передадим его в качестве callback в компонент `LoginForm`:

```
...
get_token(username, password) {
       axios.post('http://127.0.0.1:8000/api-token-auth/', {username: username,
password: password})
    .then(response => {
        console.log(response.data)
    }).catch(error => alert('Неверный логин или пароль'))
  }
```

```
  ...


  render() {
    return (
...
                        <Route exact path='/login' component={() => <LoginForm
get_token={(username, password) => this.get_token(username, password)} />} />
...
```

/frontend/src/App.js

В методе `get_token` мы отправляем методом `post` логин и пароль пользователя на адрес `/api-token-auth/`. Этот адрес мы создали на предыдущем занятии и связали с ним `View` для получения токена авторизации. Если логин и пароль верные, мы выведем ответ в консоль `console.log(response.data)`. Если нет, выведем ошибку с помощью `alert`.

```
<Route exact path='/login' component={() => <LoginForm get_token={(username,
password) => this.get_token(username, password)} />} />
```

Этот метод мы передаём в компонент `LoginForm`, чтобы вызвать его после отправки формы.

Далее в модуле Auth.js вместо вывода в консоль логина и пароля вызовем метод `get_token`:

```
handleSubmit(event) {
      this.props.get_token(this.state.login, this.state.password)
      event.preventDefault()
    }
```

/frontend/src/components/Auth.js

Теперь при вводе правильных данных пользователя в консоль выведется следующее сообщение:

```
Object { token: "b9ebacdd54a00f16cf2c2d017d0b479ce5da46b7" }
```

Это объект, содержащий ответ от сервера, в котором есть токен авторизации.

Сохранение токена авторизации

После получения токена авторизации мы уже можем прикладывать его к каждому следующему запросу. Таким образом сервер будет идентифицировать нашего пользователя.

Однако если мы закроем браузер или вкладку с нашим проектом, процедуру авторизации нужно будет проходить заново.

Чтобы запомнить токен на стороне клиента, есть два удобных способа:

- локальное хранилище браузера (localStorage);
- файл у клиента (cookies).

Рассмотрим отличия этих вариантов.

localStorage

При использовании `localStorage` данные сохраняются в кеше браузера клиента. Этот вариант хорош тем, что он удобный. Можно с помощью JavaScript сохранить данные в `localStorage` и после этого просто их извлечь.

Минус этого способа — уязвимость к XSS-атакам. Если злоумышленник сможет запустить JavaScript от имени пользователя, у него будет полный доступ к `localStorage`.

Работа с `localStorage` на JavaScript выглядит следующим образом:

```
localStorage.setItem('login', username)
let item = localStorage.getItem('login')
```

Метод `setItem` сохраняет данные, метод `getItem` получает данные из `localStorage`

Cookies

Данные хранятся в специальном файле cookies. Если при отправке cookies установить флаг httpOnly, то XSS-атаку провести не получится. Таким образом, плюс этого способа — безопасность.

Минусы: неудобство использования и размер cookies. Пользователь может запретить сохранять cookies, или данные будут слишком большого объёма.

Пример работы с cookie на JavaScript выглядит следующим образом:

```
document.cookie = `login=user;max-age=604800;domain=example.com`
```

Видно, что это менее удобно, чем работа с `localStorage`. Но если воспользоваться сторонней библиотекой, например `universal-cookie`, то работа с ними будет аналогична работе с `localStorage`:

```
import Cookies from 'universal-cookie';
const cookies = new Cookies();
cookies.set('login', 'user');
cookies.get('login')
```

В этом проекте мы будем использовать cookies, так как они безопаснее, хотя `localStorage` тоже обычно хороший вариант.

Сохранение токена авторизации

Установим библиотеку `universal-cookie` для удобной работы с cookies:

```
npm install universal-cookie
```

Изменим код в App.js следующим образом:

```
import React from 'react'
import AuthorList from './components/Author.js'
import BookList from './components/Book.js'
import AuthorBookList from './components/AuthorBook.js'
import LoginForm from './components/Auth.js'
import {BrowserRouter, Route, Switch, Redirect, Link} from 'react-router-dom'
import axios from 'axios'
import Cookies from 'universal-cookie';


const NotFound404 = ({ location }) => {
  return (
    <div>
        <h1>Страница по адресу '{location.pathname}' не найдена</h1>
    </div>
  )
}


class App extends React.Component {

  constructor(props) {
    super(props)
    this.state = {
      'authors': [],
```

```
      'books': [],
      'token': ''
    }
  }

  set_token(token) {
    const cookies = new Cookies()
    cookies.set('token', token)
    this.setState({'token': token})
  }

  is_authenticated() {
    return this.state.token != ''
  }

  logout() {
    this.set_token('')
  }

  get_token_from_storage() {
    const cookies = new Cookies()
    const token = cookies.get('token')
    this.setState({'token': token})
  }

  get_token(username, password) {
       axios.post('http://127.0.0.1:8000/api-token-auth/', {username: username,
password: password})
    .then(response => {
        this.set_token(response.data['token'])
    }).catch(error => alert('Неверный логин или пароль'))
  }

  load_data() {
    axios.get('http://127.0.0.1:8000/api/authors/')
        .then(response => {
            this.setState({authors: response.data})
        }).catch(error => console.log(error))

    axios.get('http://127.0.0.1:8000/api/books/')
        .then(response => {
            this.setState({books: response.data})
        }).catch(error => console.log(error))
  }

  componentDidMount() {
    this.get_token_from_storage()
    this.load_data()
  }

  render() {
    return (
        <div className="App">
```

```
          <BrowserRouter>
            <nav>
              <ul>
                <li>
                  <Link to='/'>Authors</Link>
                </li>
                <li>
                  <Link to='/books'>Books</Link>
                </li>
                <li>
                                                {this.is_authenticated()  ?  <button
onClick={()=>this.logout()}>Logout</button> : <Link to='/login'>Login</Link>}
                </li>
              </ul>
            </nav>
            <Switch>
                          <Route  exact  path='/'  component={()  =>  <AuthorList
items={this.state.authors} />}  />
                        <Route  exact  path='/books'  component={()  =>  <BookList
items={this.state.books} />} />
                       <Route  exact  path='/login'  component={()  =>  <LoginForm
get_token={(username, password) => this.get_token(username, password)} />} />
              <Route path="/author/:id">
                <AuthorBookList items={this.state.books} />
              </Route>
              <Redirect from='/authors' to='/' />
              <Route component={NotFound404} />
            </Switch>
          </BrowserRouter>
        </div>
    )
  }
}

export default App
```

/library/frontend/src/App.js

Рассмотрим изменения по частям.

```
import Cookies from 'universal-cookie';
```

Для работы с cookies мы будем использовать класс `Cookies` из библиотеки `universal-cookie`. С помощью него мы сможем устанавливать и читать cookies.

```
set_token(token) {
    const cookies = new Cookies()
    cookies.set('token', token)
```

```
    this.setState({'token': token})
  }
```

Метод `set_token` в классе `App` принимает токен, устанавливает его в cookies и записывает в состояние приложения. Токен в cookies нужен для сохранения пользователя при закрытии браузера, а токен в состоянии — для обновления приложения React при авторизации пользователя.

```
is_authenticated() {
    return this.state.token != ''
  }
```

Метод `is_authenticated` будет определять, авторизован пользователь или нет. Он авторизован, если токен в состоянии не пустой.

```
logout() {
    this.set_token('')
  }
```

Метод `logout` будет обнулять токен.

```
get_token_from_storage() {
    const cookies = new Cookies()
    const token = cookies.get('token')
    this.setState({'token': token})
  }
```

Метод `get_token_from_storage` нужен нам, когда мы снова открываем страницу сайта. Он считывает токен из cookies и записывает его в состояние. Таким образом при первом открытии страницы мы узнаем, был ли ранее авторизован пользователь.

После создания этих методов остаётся ещё несколько изменений:

```
get_token(username, password) {
    axios.post('http://127.0.0.1:8000/api-token-auth/', {username: username,
password: password})
    .then(response => {
        this.set_token(response.data['token'])
    }).catch(error => alert('Неверный логин или пароль'))
  }
```

После получения токена с backend вместо вывода его в консоль вызываем `this.set_token(response.data['token'])` для сохранения токена в cookies и state.

```
<li>
{this.is_authenticated() ? <button onClick={()=>this.logout()}>Logout</button> :
<Link to='/login'>Login</Link>}
                </li>
```

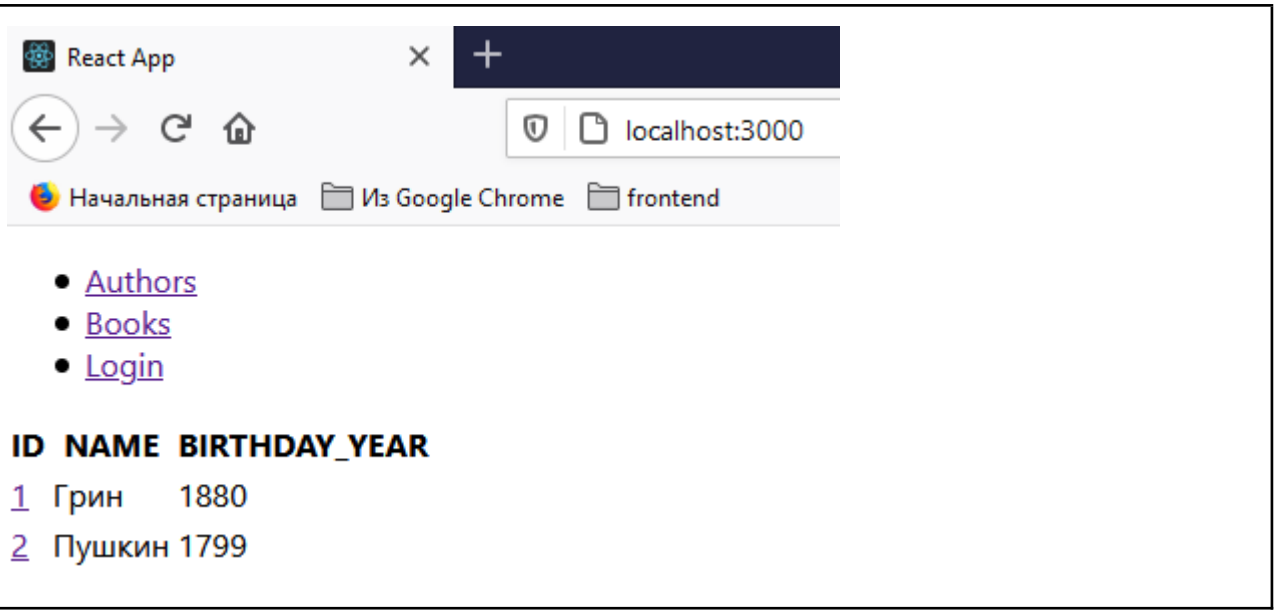
Для проверки работоспособности нашего приложения изменим пункт меню. Тут мы используем тернарный оператор. Если пользователь авторизован `this.is_authenticated()`, то рисуем кнопку Logout с обработчиком `this.logout()`, если не авторизован — ссылку на `/login`.

```
componentDidMount() {
    this.get_token_from_storage()
    this.load_data()
  }
```

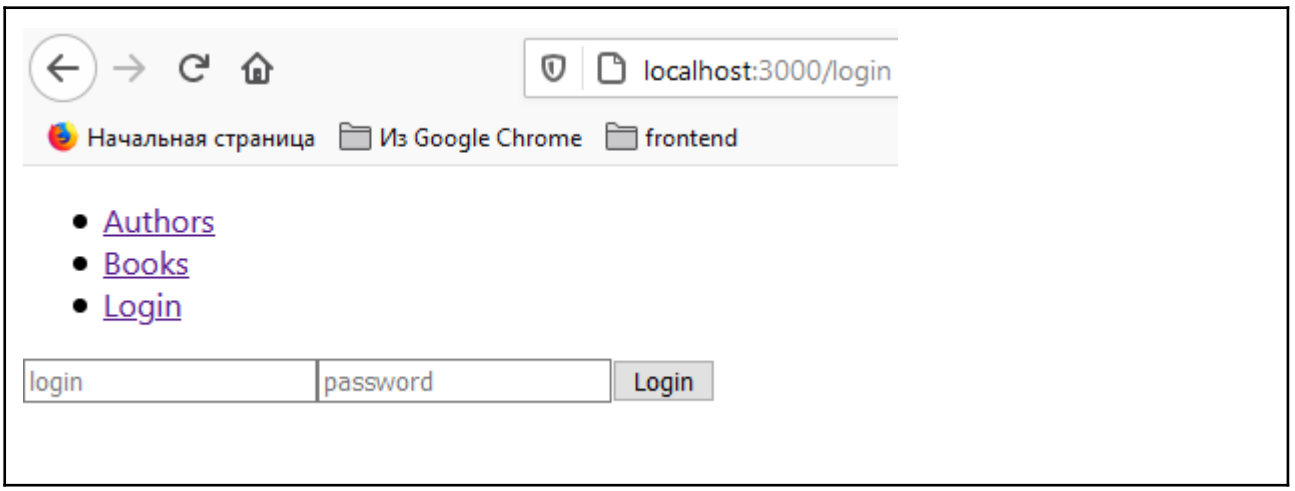
В методе `componentDidMount` вызываем `this.get_token_from_storage()`. Таким образом, если мы закроем страницу в браузере при повторном входе на сайт, мы прочитаем токен из cookies и определим, что пользователь был авторизован.

Теперь всё готово для работы. Ещё раз по шагам рассмотрим порядок работы нашего приложения.

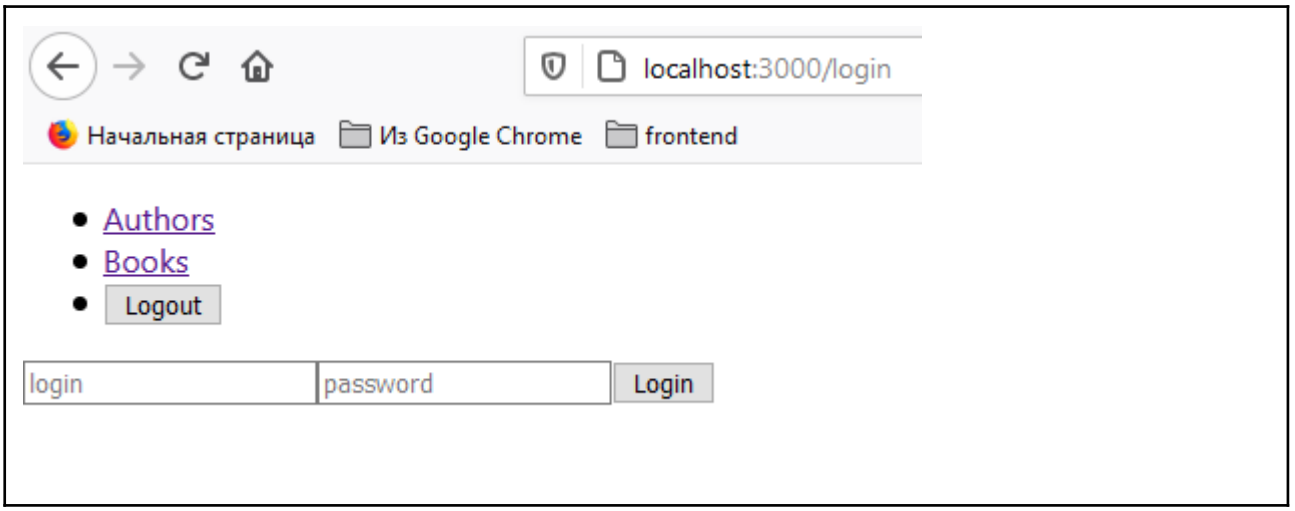
При первом входе пользователь не авторизован и видит ссылку Login:



После нажатия на ссылку Login он переходит к форме авторизации



При вводе верных логина и пароля мы получаем токен с backend, сохраняем его в cookies и обновляем state приложения. После этого ссылка Login меняется на кнопку Logout:



Далее важно проверить, сохранился ли пользователь в cookies. Для этого закрываем браузер. Снова открываем и переходим на страницу. Если всё верно, то кнопка Logout всё ещё будет активна.

Для использования `localStorage` вместо cookies достаточно изменить методы `set_token` и `get_token_from_storage`, в которых записывать токен в `localStorage` и читать из него.

Использование токена авторизации

После того как мы получили и сохранили токен авторизации, остаётся прикладывать его ко всем запросам. Это делается путём добавления токена в заголовки (headers) запроса. Это действие позволит идентифицировать пользователя на стороне сервера и задействовать систему прав.

В нашем демонстрационном проекте любой пользователь имеет права на просмотр данных, а работу по созданию данных мы рассмотрим позже. Для демонстрации работы системы прав и авторизации изменим права в `BookViewSet` на `IsAuthenticated`. Тогда для гостя просмотр данных книг будет недоступен, и мы увидим это на стороне frontend.

Изменим код файла views.py следующим образом:

```
from rest_framework import viewsets, permissions
from .models import Author, Book
from .serializers import AuthorSerializer, BookSerializer


class AuthorViewSet(viewsets.ModelViewSet):
    serializer_class = AuthorSerializer
    queryset = Author.objects.all()


class BookViewSet(viewsets.ModelViewSet):
    permission_classes = [permissions.IsAuthenticated]
    serializer_class = BookSerializer
    queryset = Book.objects.all()
```

/library/mainapp/views.py

Мы добавили `permisson_classes` в класс `BookViewSet`.

Теперь выйдем (нажмём кнопку Logout) и обновим страницу на клиенте. Мы увидим, что данные книг не загрузились. В консоли на клиенте мы получим:

```
Error: Request failed with status code 401
```

А в консоли сервера:

```
Unauthorized: /api/books/
```

Если повторить попытку авторизованным пользователем, будет то же самое, так как мы не отправляем токен авторизации в заголовках запроса.

Передача токена в заголовках запроса

Изменим код в файле App.js следующим образом:

```
import React from 'react'
import AuthorList from './components/Author.js'
import BookList from './components/Book.js'
import AuthorBookList from './components/AuthorBook.js'
import LoginForm from './components/Auth.js'
import {BrowserRouter, Route, Switch, Redirect, Link} from 'react-router-dom'
import axios from 'axios'
import Cookies from 'universal-cookie';


const NotFound404 = ({ location }) => {
  return (
    <div>
        <h1>Страница по адресу '{location.pathname}' не найдена</h1>
    </div>
  )
}


class App extends React.Component {

  constructor(props) {
    super(props)
    this.state = {
      'authors': [],
      'books': [],
      'token': ''
    }
  }

  set_token(token) {
    const cookies = new Cookies()
    cookies.set('token', token)
    this.setState({'token': token}, ()=>this.load_data())
  }

  is_authenticated() {
    return this.state.token != ''
  }

  logout() {
    this.set_token('')
  }

  get_token_from_storage() {
    const cookies = new Cookies()
    const token = cookies.get('token')
    this.setState({'token': token}, ()=>this.load_data())
```

```
  }

  get_token(username, password) {
       axios.post('http://127.0.0.1:8000/api-token-auth/', {username: username,
password: password})
    .then(response => {
        this.set_token(response.data['token'])
    }).catch(error => alert('Неверный логин или пароль'))
  }

  get_headers() {
    let headers = {
      'Content-Type': 'application/json'
    }
  if (this.is_authenticated())
    {
        headers['Authorization'] = 'Token ' + this.state.token
    }
    return headers
  }


  load_data() {

    const headers = this.get_headers()
    axios.get('http://127.0.0.1:8000/api/authors/', {headers})
        .then(response => {
            this.setState({authors: response.data})
        }).catch(error => console.log(error))

    axios.get('http://127.0.0.1:8000/api/books/', {headers})
        .then(response => {
            this.setState({books: response.data})
        }).catch(error => {
          console.log(error)
          this.setState({books: []})
        })
  }

  componentDidMount() {
    this.get_token_from_storage()
  }

  render() {
    return (
        <div className="App">
          <BrowserRouter>
            <nav>
              <ul>
                <li>
                  <Link to='/'>Authors</Link>
                </li>
                <li>
```

```
                <Link to='/books'>Books</Link>
              </li>
              <li>
                                                {this.is_authenticated()  ?  <button
onClick={()=>this.logout()}>Logout</button> : <Link to='/login'>Login</Link>}
              </li>
            </ul>
          </nav>
          <Switch>
                        <Route  exact  path='/'  component={()  =>  <AuthorList
items={this.state.authors} />}  />
                      <Route  exact  path='/books'  component={()  =>  <BookList
items={this.state.books} />} />
                     <Route  exact  path='/login'  component={()  =>  <LoginForm
get_token={(username, password) => this.get_token(username, password)} />} />
            <Route path="/author/:id">
              <AuthorBookList items={this.state.books} />
            </Route>
            <Redirect from='/authors' to='/' />
            <Route component={NotFound404} />
          </Switch>
        </BrowserRouter>
      </div>
    )
  }
}

export default App
```

/library/frontend/src/App.js

Рассмотрим внесённые изменения по частям:

```
 get_headers() {
    let headers = {
      'Content-Type': 'application/json'
    }
  if (this.is_authenticated())
    {
        headers['Authorization'] = 'Token ' + this.state.token
    }
    return headers
  }
```

Мы добавили метод `get_headers`. Если пользователь авторизован, то в заголовки запроса мы добавляем ключ `Authorization` со значением нашего токена авторизации:

```
  load_data() {
```

```
    const headers = this.get_headers()
    axios.get('http://127.0.0.1:8000/api/authors/', {headers})
        .then(response => {
            this.setState({authors: response.data})
        }).catch(error => console.log(error))

    axios.get('http://127.0.0.1:8000/api/books/', {headers})
        .then(response => {
            this.setState({books: response.data})
        }).catch(error => {
          console.log(error)
          this.setState({books: []})
        })
  }
```

В методе `load_data` мы получаем заголовки запроса и прикладываем их к каждому запросу.

Далее остаётся несколько небольших, но важных деталей.

```
set_token(token) {
    const cookies = new Cookies()
    cookies.set('token', token)
    this.setState({'token': token}, ()=>this.load_data())
  }
...
get_token_from_storage() {
    const cookies = new Cookies()
    const token = cookies.get('token')
    this.setState({'token': token}, ()=>this.load_data())
  }
```

В методах `get_token` и `get_token_from_storage` при вызове `this.setState` мы добавили второй параметр. Это callback на вызов функции `this.load_data`. Это нужно потому, что изменение состояния происходит асинхронно. Если не указать callback, который будет срабатывать сразу после изменения состояния, то данные загрузятся раньше, чем изменится состояние `this.state.token`.

```
 componentDidMount() {
    this.get_token_from_storage()
  }
```

В `componentDidMount` больше не нужно вызывать `this.load_data()`. Этот метод сработает после загрузки токена из cookies.

```
axios.get('http://127.0.0.1:8000/api/books/', {headers})
```

```
        .then(response => {
            this.setState({books: response.data})
        }).catch(error => {
          console.log(error)
          this.setState({books: []})
        })
```

При загрузке книг в блоке `catch` мы добавили `this.setState({books: []})`. Это нужно для случая, когда мы сделаем logout. Запрос при отправке на сервер будет без заголовков, и сервер вернёт ошибку. Но при этом данные о книгах останутся в состоянии `state.books`. Для этого мы их обнуляем.

Всё готово для проверки работоспособности нашего проекта.

Теперь для гостя данные о книгах не загружаются. Они загружаются как после логина, так и если мы закрыли и открыли вкладку браузера. Если пользователь вышел, то данные снова становятся пустыми.

Итоги

На этом занятии мы рассмотрели процесс авторизации на стороне клиента, чтобы понять, как работает система прав и авторизации в DRF.

Глоссарий

Куки (англ. cookie, «печенье») — небольшой фрагмент данных, отправленный веб-сервером и хранимый на компьютере пользователя. Веб-клиент (обычно веб-браузер) всякий раз при попытке открыть страницу соответствующего сайта пересылает этот фрагмент данных веб-серверу в составе HTTP-запроса. Применяется для сохранения данных на стороне пользователя. На практике обычно используется для:

- аутентификации пользователя;
- хранения персональных предпочтений и настроек пользователя;
- отслеживания состояния сеанса доступа пользователя;
- сведения статистики о пользователях.

LocalStorage — хранилище данных в браузере клиента

Дополнительные материалы

1. localStorage.
2. universal-cookie.
3. cookie vs localStorage (статья).

Используемые источники

1. localStorage.
2. universal-cookie.
3. cookie vs localStorage (статья).
4. Про токены и авторизацию.

Практическое задание

Добавить авторизацию на стороне клиента.

В этой самостоятельной работе мы тренируем умения:

- получать токен авторизации;
- хранить токен авторизации;
- прикладывать токен авторизации к запросу на сервер.

Смысл: использовать авторизацию на стороне клиента в React. Лучше понимать механизмы авторизации и системы прав

Последовательность действий

1. Создать компонент для авторизации с формой логина и пароля пользователя.
2. При отправке формы получить токен пользователя.
3. Сохранить токен пользователя в localStorage или cookies.
4. Добавить кнопку logout («выйти»). По нажатию на неё очищать токен в localStorage или cookies.
5. Прикладывать токен к последующим запросам.
6. * На всех страницах отображать имя авторизованного пользователя и кнопку «Выйти», либо кнопку «Войти», если пользователь не авторизован.