Лекция 8 - Управление состоянием приложения, работа с сетью

Сегодняшняя наша тема - управление состоянием приложения и работа с сетью, включая использование REST API и библиотек для сетевого взаимодействия. Эти аспекты являются ключевыми для создания современных интерактивных приложений, будь то веб-сайты или мобильные приложения.

Управление состоянием приложения

Управление состоянием - это процесс отслеживания изменений, которые происходят с данными в вашем приложении. Это может включать пользовательский ввод, данные, полученные от сервера, пользовательские предпочтения и так далее.

Зачем нужно управлять состоянием?

- Повышение производительности: Корректное управление избегает ненужных обновлений и перерисовок.
- Легкость отладки: Состояние приложения можно инспектировать и изменять на лету.
- Предсказуемость поведения: Зная текущее состояние, можно точно предсказать реакцию приложения на действия пользователя.
- Масштабируемость: Правильное управление состоянием упрощает добавление новых функций и возможностей.

Паттерны управления состоянием:

- Global State Management (Redux, Vuex, NgRx)
- Local State Management (useState в React, data в Vue)
- Context API / Provide & inject механизмы для избегания "prop drilling".
- State Management Libraries (MobX, Recoil)

Работа с сетью

REST API

REST (Representational State Transfer) - это архитектурный стиль взаимодействия компонентов распределенного приложения в сети. RESTful API - это API, который следует принципам REST.

Компоненты REST:

- Ресурсы: Основная абстракция информации в REST, обычно представляется в виде URI.
- HTTP методы: GET, POST, PUT, DELETE и др. определяют действия над ресурсами.
- Статусы ответов HTTP: Коды (200, 404, 500 и т.д.), указывающие на результат операции.

JSON

JSON (JavaScript Object Notation) - это текстовый формат обмена данными, который легко читается как человеками, так и машинами.

Библиотеки для сетевого взаимодействия:

- Fetch API: Встроенный в браузеры инструмент для работы с HTTP-запросами.

- Axios: Популярная библиотека JavaScript, которая упрощает выполнение HTTP-запросов.
- Retrofit для Android: Типичный выбор для взаимодействия с REST API на Android.
- Alamofire для iOS: Мощная библиотека для работы с HTTP-запросами на Swift.

Примеры кода

Давайте рассмотрим простой пример с использованием Axios для отправки GET-запроса:

Javascript:

```
import axios from 'axios';

axios.get('https://api.example.com/data')
  .then(response => {
    console.log(response.data);
  })
  .catch(error => {
    console.error('Ошибка при выполнении запроса:', error);
  });
```

А теперь пример с использованием Fetch API:

```
fetch('https://api.example.com/data')
  .then(response => {
    if (!response.ok) {
      throw new Error('Network response was not ok');
    }
    return response.json();
  })
  .then(data => {
    console.log(data);
  })
  .catch(error => {
    console.error('Ошибка при выполнении запроса:', error);
  });
```

Пример управления состоянием в React с использованием Redux

Для начала посмотрим, как можно использовать библиотеку Redux для управления состоянием в приложении React:

```
// action.js
export const LOGIN_SUCCESS = 'LOGIN_SUCCESS';

export function loginSuccess(userData) {
  return { type: LOGIN_SUCCESS, payload: userData };
}

// reducer.js
```

```
import { LOGIN_SUCCESS } from './actions';

const initialState = {
  user: null,
};

function authReducer(state = initialState, action) {
  switch (action.type) {
    case LOGIN_SUCCESS:
      return { ...state, user: action.payload };
    default:
      return state;
  }
}

export default authReducer;

// store.js
import { createStore } from 'redux';
import authReducer from './reducer';

const store = createStore(authReducer);

export default store;

// App.js
import React from 'react';
import { useSelector, useDispatch } from 'react-redux';
import { loginSuccess } from './actions';

function App() {
  const user = useSelector(state => state.user);
  const dispatch = useDispatch();

  function handleLogin() {
    const userData = { name: 'Alice' }; // Предположим, данные получены после аутентификации
    dispatch(loginSuccess(userData));
  }

  return (
    <div>
      {user ? (
        <h1>Добро пожаловать, {user.name}!</h1>
      ) : (
        <button onClick={handleLogin}>Войти</button>
      )}
    </div>
  );
}
```

```
export default App;
```

Пример работы с REST API с использованием async/await в JavaScript

```
async function getUserData(userId) {
  try {
    const response = await fetch(`https://api.example.com/users/${userId}`);
    if (!response.ok) {
      throw new Error(`Ошибка: ${response.status}`);
    }
    const data = await response.json();
    console.log('Полученные данные пользователя:', data);
  } catch (error) {
    console.error('Произошла ошибка при получении данных пользователя:', error);
  }
}

getUserData(1);
```

Пример POST-запроса с Axios

```
import axios from 'axios';

async function createUser(userData) {
  try {
    const response = await axios.post('https://api.example.com/users', userData);
    console.log('Пользователь создан:', response.data);
  } catch (error) {
    console.error('Ошибка при создании пользователя:', error);
  }
}

const newUser = {
  name: 'John Doe',
  email: 'john.doe@example.com',
};

createUser(newUser);
```

Эти примеры демонстрируют основные концепции управления состоянием и взаимодействия с сетью в современных JavaScript-приложениях. Примеры кода представлены в сокращенной форме и предполагают базовое понимание соответствующих фреймворков и библиотек.

Управление состоянием и работа с сетью - это фундаментальные аспекты современной разработки приложений. Использование REST API и библиотек для сетевого взаимодействия позволяет приложениям быть интерактивными и динамичными, обмениваясь данными с сервером и предоставляя пользователям

актуальную информацию. Важно понимать принципы работы с состоянием и сетевыми запросами для создания эффективных и масштабируемых приложений. Спасибо за внимание, теперь перейдем к вопросам и практическим заданиям.