Приднестровский государственный университет им. Т.Г. Шевченко
Физико-технический институт

ОСНОВЫ ВЕБ-РАЗРАБОТКИ С ПРИМЕНЕНИЕМ ИИ-АССИСТЕНТОВ

Лабораторный практикум

Разработал:
ст. преподаватель
кафедры ИТ
Бричаг Д.В.

г. Тирасполь
2025 г.

Лабораторная работа №9

Работа с внешними данными (API + React Hooks)

Цель работы:

- Научиться подключаться к внешним API в React.
- Освоить хук **useEffect** для выполнения побочных эффектов.
- Понять логику загрузки данных, обработки ошибок и отображения состояния загрузки.
- Использовать ИИ для ускорения разработки и сравнения решений.
- Создать **мини-React-приложение**, работающее с реальными данными.

Теоретическая справка

Проблема: Почему нельзя просто написать fetch?

В обычном JavaScript (или jQuery) вы могли бы написать код так:

```
// ❌ ТАК ДЕЛАТЬ В REACT НЕЛЬЗЯ
function UserProfile() {
  // Пытаемся скачать данные прямо в теле функции
  fetch('/api/user').then(data => ...);

  return <div>Профиль...</div>;
}
```

Почему это катастрофа? React-компонент — это функция. Она вызывается каждый раз, когда компонент нужно перерисовать (рендер).

1. Компонент рендерится.
2. Запускается fetch.
3. Когда fetch получает данные, мы обычно обновляем состояние (useState).
4. Обновление состояния вызывает новый рендер.
5. Компонент снова рендерится -> Снова запускается fetch...

Результат: Бесконечный цикл запросов, который "положит" ваш браузер или забанит вас на API.

Решение: Хук useEffect (Побочные эффекты)

useEffect — это специальное место для кода, который должен выполняться не во время рисования, а после него, и при определенных условиях.

Это идеальное место для "побочных эффектов" (Side Effects): запросов к API, таймеров, подписок, изменения заголовка документа.

Синтаксис:

```
useEffect(() => {
  // Код эффекта (выполнится после рендера)
}, [ /* Массив зависимостей */ ]);
```

Магия массива зависимостей (Dependency Array)

Второй аргумент useEffect — массив [] — работает как фильтр:

1. [] (Пустой массив): Эффект выполнится только один раз после первого рендера (аналог "при загрузке страницы"). Идеально для API.

2. [id] (Массив с переменными): Эффект выполнится при первом рендере И каждый раз, когда меняется переменная id.

- *Пример:* Пользователь переключился с профиля id=1 на id=2 -> React видит изменение и перезапускает запрос.

3. Нет массива (пропущен): Эффект выполняется после каждого рендера. (Опасно для API!).

«Святая троица» состояния загрузки

В отличие от jQuery, где мы просто меняли DOM по завершении запроса, в React интерфейс должен отражать состояние данных в любой момент времени.

При работе с сетью у нас всегда есть 3 состояния:

1. Loading (Загрузка): Данных еще нет, крутим спиннер.

2. Success (Успех): Данные пришли, показываем контент.

3. Error (Ошибка): Сервер упал или интернет пропал, показываем сообщение.

Мы должны создать useState для каждого из этих состояний (или один объект).

4. Правильный паттерн (Best Practice)

Вот как выглядит профессиональный код для загрузки данных. Обратите внимание на создание асинхронной функции внутри эффекта.

Важно: Саму функцию useEffect нельзя делать async (useEffect(async () => ...) – ошибка). React ожидает, что эффект вернет функцию очистки или ничего, а async функция возвращает Promise.

```
import { useState, useEffect } from 'react';

function UserList() {
  // 1. Объявляем состояния
  const [users, setUsers] = useState([]);        // Сами данные
  const [isLoading, setIsLoading] = useState(true); // Индикатор загрузки
  const [error, setError] = useState(null);      // Текст ошибки

  useEffect(() => {
    // 2. Создаем функцию загрузки внутри эффекта
    const fetchData = async () => {
      try {
        // Сбрасываем ошибку перед новой попыткой (если нужно)
        setError(null);

        const response = await fetch('https://jsonplaceholder.typicode.com/users');

        // fetch не считает 404 ошибкой, проверяем вручную
        if (!response.ok) {
          throw new Error(`Ошибка HTTP: ${response.status}`);
        }

        const data = await response.json();
        setUsers(data); // Сохраняем данные
      } catch (err) {
        setError(err.message); // Ловим ошибки сети или парсинга
      } finally {
        setIsLoading(false); // Убираем спиннер в ЛЮБОМ случае
      }
    };

    fetchData(); // 3. Вызываем её
  }, []); // 4. Пустой массив = запускаем 1 раз при монтировании

  // 5. Условный рендеринг (Conditional Rendering)
  if (isLoading) return <div className="spinner">Загрузка...</div>;
  if (error) return <div className="error">Ошибка: {error}</div>;

  return (
    <ul>
```

```
        {users.map(user => (
          <li key={user.id}>{user.name}</li>
        ))}
      </ul>
    );
  }
```

Практическая часть

Мини-проекты:

Вариант A – "Каталог фильмов" (Movie Search)

API: https://www.omdbapi.com/

Вариант B – "Галерея котиков по породам"

API: https://thecatapi.com/

Вариант C – "GitHub User Finder"

API: https://api.github.com/users/username

Общие шаги для всех вариантов

Шаг 1. Подготовка окружения

Создать новый React-проект:

```
npm create vite@latest lab9-react-api --template react
cd lab9-react-api
npm install
npm run dev
```

Шаг 2. Использование ИИ

Примеры промптов, которые вы можете использовать для создания скелета своего приложения:

```
Сделай React-компонент, который получает данные из <API> и выводит список элементов.
Используй useEffect и async/await.
```

```
Добавь индикатор загрузки и обработку ошибок.
```

```
Объясни, почему useEffect не должен быть async, и как правильно внутри него делать
асинхронный код.
```

```
Перепиши этот код, чтобы он был безопаснее и короче.
```

Вариант A — Каталог фильмов

Функционал:

- Поле ввода названия фильма
- По клику — запрос к API
- Отображение списка найденных фильмов
- Модалка с подробностями по конкретному фильму
- Состояния: *loading*, *error*, *movies*, *selectedMovie*

Минимальная структура:

```
components/
  SearchBar.jsx
  MovieList.jsx
  MovieCard.jsx
  MovieModal.jsx
App.jsx
```

Пример API-запроса:

```
const res = await fetch(`https://www.omdbapi.com/?apikey=demo&s=${query}`);
```

Вариант B — Каталог котиков по породам

Функционал:

- Dropdown выбора породы
- useEffect → загрузка списка пород
- Выбор породы → загрузка фото
- Грид картинок
- Ошибки, состояние загрузки

Пример API-запроса:

```
fetch("https://api.thecatapi.com/v1/breeds")
```

Вариант C — GitHub User Finder

Функционал:

- Поле ввода логина GitHub
- Загрузка профиля:
  - аватар
  - количество репозиториев
  - ссылка на профиль
- Кнопка “загрузить репозитории”
- Список репозиториев

Пример API-запроса:

```
fetch(`https://api.github.com/users/${username}`)
```

Шаг 3. Обязательный функционал для всех мини-апок

Каждое приложение должно включать:

1. Загрузка данных через API

fetch или axios — студент может выбрать.

2. useEffect

Эффект должен:

- вызываться один раз при загрузке страницы

  или

- вызываться при изменении поискового запроса.

3. Обработка ошибок

```
catch(err => setError("Ошибка загрузки данных"))
```

4. Индикация загрузки

```
Loading…
```

5. Минимальный UI

Грид (сетка) карточек, список элементов – на ваше усмотрение. Не бойтесь использовать свою фантазию и возможности нейросетей.

6. Использование ИИ

LLM помогает:

- написать запрос
- оптимизировать код
- объяснить зависимости useEffect

Работа с git и GitHub

1. Инициализируйте репозиторий:

```
git init
git add .
git commit -m "Lab9: React [theme] App"
```

2. Создайте новый репозиторий на GitHub.

3. Свяжите локальный проект с GitHub:

```
git remote add origin https://github.com/<user>/lab9-react-api.git
git branch -M main
git push -u origin main
```

Отчёт (в README.md)

В README.md добавить:

- Название выбранного мини-проекта
- Пример использованного API-запроса
- Скриншоты работы приложения
- Ответы на вопросы:

- Что делает useEffect в вашем приложении?
- Какие состояния вы использовали и зачем?
- Где ИИ помог, а где пришлось разбираться самому?
- Что из документации API было важнее всего?

Результаты работы

В итоге у вас должно быть:

1. Рабочее React-приложение.
2. Исходный код с компонентами и состоянием.
3. Доработан функционал согласно чек-листу методических материалов.
4. Репозиторий на GitHub и отчёт в README.md.

Критерии оценки

- Приложение запускается и работает (2 балла).
- Работает загрузка данных из API (2 балла).
- Есть useEffect и состояние loading/error/data (2 балла).
- Реализован функционал выбора / поиска / фильтрации (2 балла).
- Код выложен на GitHub и оформлен отчёт (2 балла).