

**Лабораторная работа № 9**

**Асинхронные запросы   
(Fetch API, AJAX)**

Студент: Маркелов Дмитрий

ИС 24-21

**Контрольные вопросы:**

1. ***Современные технологии для информационных систем:*** *облачные вычисления, искусственный интеллект, большие данные (Big Data), Интернет вещей (IoT), блокчейн, виртуальная и дополненная реальность.*
2. ***Суть облачных вычислений:*** *предоставление вычислительных ресурсов (серверы, хранилища, приложения) через интернет по запросу, без необходимости локальной установки.*
3. ***Преимущества и риски ИИ:***
   * *Преимущества: автоматизация, анализ больших данных, персонализация, повышение эффективности.*
   * *Риски: ошибки алгоритмов, утрата рабочих мест, этические проблемы, возможное нарушение приватности.*
4. ***Влияние Интернета вещей:*** *умные дома, носимые устройства, автоматизация транспорта и городских систем, улучшение мониторинга здоровья и ресурсов.*
5. ***Почему блокчейн — «технология доверия»:*** *данные хранятся децентрализованно и защищены криптографически, изменения прозрачны и практически невозможны без согласия сети.*
6. ***Аспекты безопасности ИТ в образовании и науке:*** *защита персональных данных, контроль доступа, защита от кибератак, безопасное хранение и передача научной информации.*

**📚 Краткий теоретический обзор Fetch API**

**Fetch API** — это современный способ выполнения асинхронных HTTP-запросов в JavaScript.  
Он заменяет устаревший механизм **XMLHttpRequest**, обеспечивая более простой и удобный синтаксис.

Основной принцип работы Fetch API — использование **промисов**.  
Пример простого запроса:

fetch('https://example.com/data')

.then(response => response.json())

.then(data => console.log(data))

.catch(error => console.error('Ошибка:', error));

Метод fetch() возвращает объект **Promise**, который:

* Переходит в состояние *fulfilled* (успешно выполнен), когда сервер ответил.
* Переходит в состояние *rejected*, если произошла ошибка сети или запрос не удался.

Чтобы обрабатывать ошибки, нужно проверять свойство response.ok.  
Для удобства часто используется async/await, что делает код похожим на синхронный.

### Листинг программы (HTML + JS)

**Файл:** index.html

<!DOCTYPE html>

<html lang="ru">

<head>

<meta charset="UTF-8">

<title>Список пользователей</title>

<style>

body {

font-family: Arial, sans-serif;

margin: 20px

}

h1 {

text-align: center;

}

#loading {

display: none;

font-weight: bold;

text-align: center;

margin: 20px 0;

}

#error {

color: red;

text-align: center;

margin: 20px 0;

display: none;

}

table {

width: 80%;

margin: 0 auto;

border-collapse: collapse;

}

th, td {

border: 1px solid #aaa;

padding: 8px;

text-align: left;

}

th {

background-color: #f2f2f2;

}

button {

display: block;

margin: 20px auto;

padding: 10px 20px;

font-size: 16px;

}

</style>

</head>

<body>

<h1>Список пользователей</h1>

<div id="loading">Загрузка...</div>

<div id="error">Ошибка при получении данных</div>

<table id="usersTable">

<thead>

<tr>

<th>ID</th>

<th>Имя пользователя</th>

<th>Email</th>

<th>Город</th>

</tr>

</thead>

<tbody>

<!-- Данные пользователей будут добавлены здесь -->

</tbody>

</table>

<button id="refreshBtn">Обновить данные</button>

<script>

const usersTableBody = document.querySelector('#usersTable tbody');

const loadingDiv = document.getElementById('loading');

const errorDiv = document.getElementById('error');

const refreshBtn = document.getElementById('refreshBtn');

// Функция загрузки пользователей

async function loadUsers() {

usersTableBody.innerHTML = '';

errorDiv.style.display = 'none';

loadingDiv.style.display = 'block';

try {

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

if (!response.ok) {

throw new Error('Сетевая ошибка');

}

const users = await response.json();

users.forEach(user => {

const row = document.createElement('tr');

row.innerHTML = `

<td>${user.id}</td>

<td>${user.username}</td>

<td>${user.email}</td>

<td>${user.address.city}</td>

`;

usersTableBody.appendChild(row);

});

} catch (error) {

errorDiv.style.display = 'block';

console.error('Ошибка при получении данных:', error);

} finally {

loadingDiv.style.display = 'none';

}

}

// Загрузка данных при запуске

loadUsers();

// Обновление данных по кнопке

refreshBtn.addEventListener('click', () => {

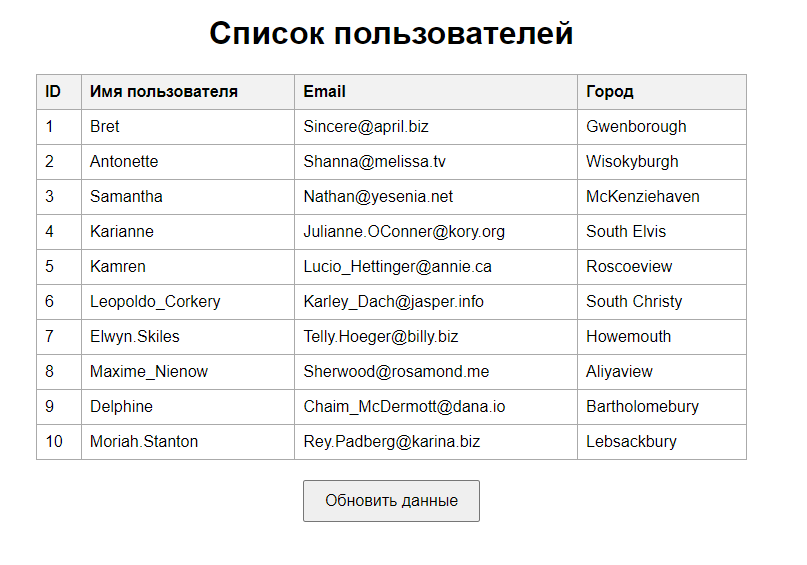
loadUsers();

});

</script>

</body>

</html>



### 📈 Выводы

В ходе работы было освоено:

* использование **Fetch API** для выполнения асинхронных HTTP-запросов;
* работа с **Promise** и конструкцией **async/await**;
* обработка ошибок при сетевых запросах;
* динамическое создание HTML-элементов (таблицы) на JavaScript;
* реализация простого интерфейса с кнопкой обновления данных и индикатором загрузки.

**Возникшие трудности и решения:**

* При первой попытке данные не отображались — решено добавлением await response.json().
* Добавлена обработка ошибок через try...catch для вывода сообщений пользователю.

**Итог:**  
Цель работы достигнута. Получены практические навыки работы с Fetch API и асинхронным JavaScript.