

## **ПМЗ Разработка модулей ПО.**

**РО 3.1 Понимать и применять принципы объектно-ориентированного и асинхронного программирования.**

# **Тема 2. Асинхронно программирование.**

## **Лекция 13. Асинхронные операции в реальности: HTTP-запросы.**

# Цель занятия:

Показать, как работают асинхронные HTTP-запросы в JavaScript, разобрать инструменты **fetch** и **XMLHttpRequest**, их связь с промисами и использование в реальных приложениях.

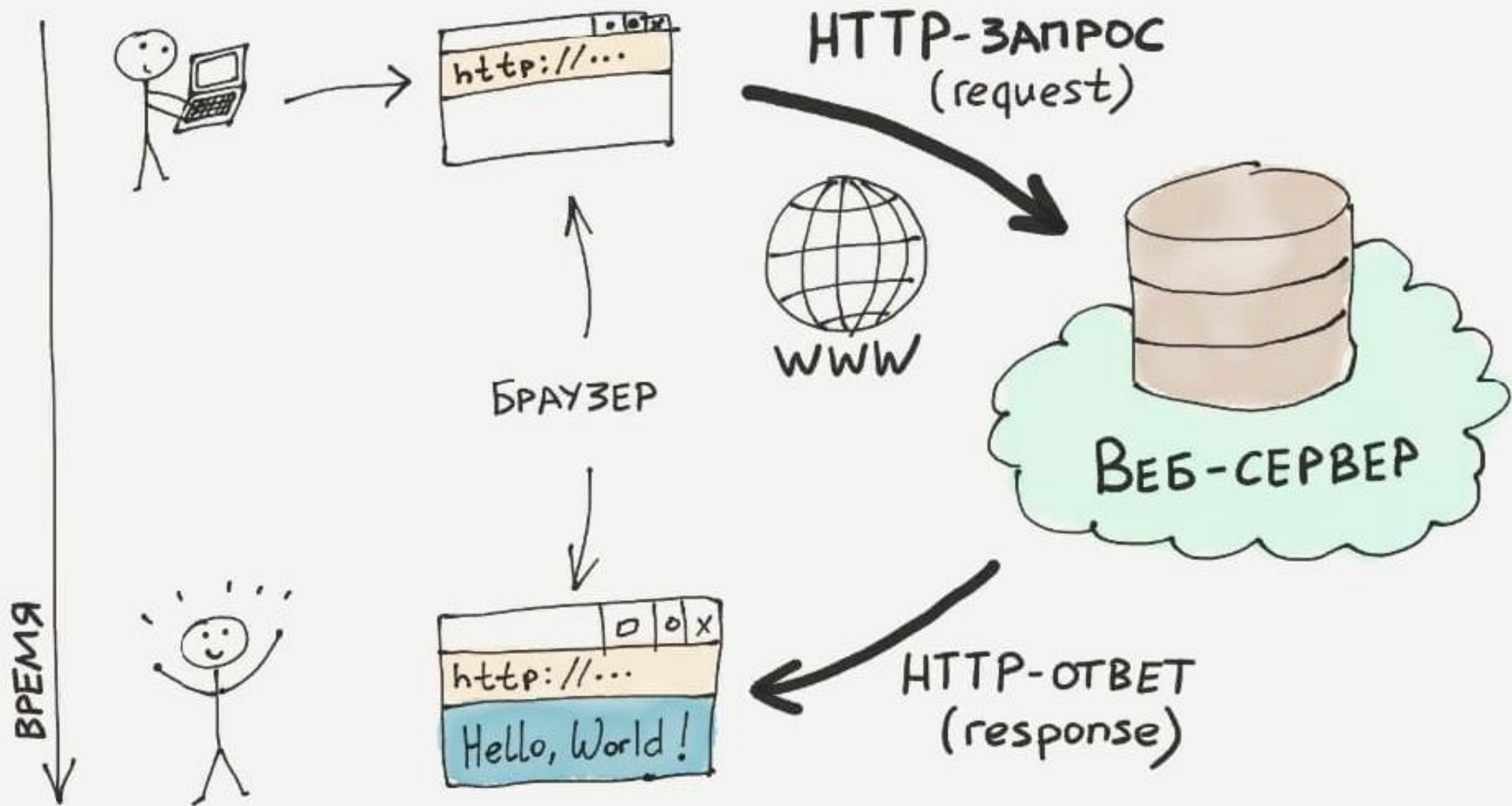
# Учебные вопросы:

1. Что такое HTTP-запрос и зачем он нужен в JS.
2. История: XMLHttpRequest и его особенности.
3. Современный подход: метод fetch.
4. fetch и промисы.
5. Использование async/await для HTTP-запросов
6. Обработка ошибок: сетевые ошибки и статус ответа.

# 1. Что такое HTTP-запрос и зачем он нужен в JS.

HTTP-запрос — это обращение клиента (обычно браузера) к серверу по протоколу HTTP (HyperText Transfer Protocol) с целью получить или отправить данные.

В JavaScript HTTP-запросы используются для обмена информацией между веб-страницей и сервером без перезагрузки страницы.




## Роль в веб-разработке.

- Позволяют загружать данные (например, список товаров, новости, посты).
- Дают возможность отправлять данные на сервер (формы, сообщения, результаты работы пользователя).
- Используются для связи с API (Application Programming Interface) — наборами готовых функций сервера, которые возвращают данные в формате JSON или XML.
- Являются основой для динамических приложений (SPA, чаты, онлайн-магазины, соцсети).

## Как это связано с JavaScript?

JavaScript может делать асинхронные запросы к серверу:

- запрос отправляется,
- JS продолжает выполнять другие задачи,
- когда сервер отвечает, результат обрабатывается (через колбэки, промисы или `async/await`).

 Это позволяет создавать интерфейсы, которые не блокируются во время ожидания ответа от сервера.



## Пример реальной задачи.

- Пользователь открывает интернет-магазин.
- Страница не содержит всех данных сразу.
- JavaScript выполняет HTTP-запрос к серверу: «дай список товаров».
- Сервер возвращает JSON с товарами.
- JS строит список товаров прямо в браузере без перезагрузки страницы.

Ключевые методы для работы с HTTP-запросами в JS:

- XMLHttpRequest — **старый способ**, появился ещё до появления промисов.
- fetch — **современный метод**, основанный на промисах, удобнее и проще в использовании.

### **Вывод:**


- HTTP-запросы — это механизм взаимодействия браузера с сервером.
- В JavaScript они позволяют создавать динамичные веб-приложения: получать и отправлять данные асинхронно, не перезагружая страницу.

## 2. История: XMLHttpRequest и его особенности.

### Что такое XMLHttpRequest?

XMLHttpRequest — это встроенный объект в JavaScript, с помощью которого можно:

- отправлять запросы на сервер,
- получать ответы от сервера,
- работать асинхронно (в фоне).

 Несмотря на название, XHR работает не только с XML, а чаще всего — с JSON (данные в виде обычного текста, похожего на JavaScript-объекты).

## Простейший запрос выглядит так:

```
let xhr = new XMLHttpRequest(); // 1. Создаём объект

xhr.open("GET", "https://jsonplaceholder.typicode.com/posts/1");
// 2. Говорим: хотим сделать запрос "GET" по адресу (URL)

xhr.send();
// 3. Отправляем запрос

xhr.onload = function() { // 4. Ждём, когда придёт ответ
  if (xhr.status === 200) { // Проверяем, что запрос прошёл успешно
    console.log("Ответ сервера:", xhr.responseText);
  } else {
    console.error("Ошибка:", xhr.status);
  }
};
```

## Результат:

```
Ответ сервера: { test.js:12  
  "userId": 1,  
  "id": 1,  
  "title": "sunt aut facere repellat provident occaecati excepturi  
optio reprehenderit",  
  "body": "quia et suscipit\nsuscipit recusandae consequuntur  
expedita et cum\nreprehenderit molestiae ut ut quas totam\nnostrum  
rerum est autem sunt rem eveniet architecto"  
}
```

>

## Пояснение к `xhr.open("GET", "...")`

Метод `open()` настраивает запрос:

- Первый параметр — метод HTTP (например, "GET", "POST" и др.).
- Второй параметр — адрес (URL), куда мы отправляем запрос.

### В примере:

```
xhr.open("GET", "https://jsonplaceholder.typicode.com/posts/1");
```

- "GET" означает: «получи данные с сервера».
- Адрес "https://jsonplaceholder.typicode.com/posts/1" — это тестовый сервер, который вернёт один пост в формате JSON.

## Кратко про методы HTTP:

- **GET** — получить данные (самый частый вариант).
- **POST** — отправить новые данные на сервер.
- **PUT** — обновить существующие данные.
- **DELETE** — удалить данные.

## Особенности XMLHttpRequest:

- Поддерживается всеми браузерами.
- Можно использовать события (onload, onerror, onprogress).
- Код получается более «тяжёлым», чем с fetch.



### Вывод:

- XMLHttpRequest — **старый способ** работы с HTTP-запросами, который дал начало AJAX.
- Он **до сих пор работает**, но для новых проектов удобнее использовать fetch.



# 3. Современный подход: метод `fetch`.

## Что такое `fetch`?

- `fetch` — это встроенный в браузеры (и новые версии Node.js) метод для выполнения HTTP-запросов.
- Он заменяет устаревший `XMLHttpRequest` и основан на промисах.
- Удобнее и проще для чтения кода.

## Синтаксис:

```
fetch(url, options)
  .then(response => {
    // ответ от сервера
  })
  .catch(error => {
    // ошибка сети
  });
```

- url — адрес, куда отправляется запрос.
- options — объект с настройками (метод, заголовки, тело запроса). По умолчанию используется метод GET.

## Простейший пример (GET-запрос):

```
fetch("https://jsonplaceholder.typicode.com/posts/1")  
  .then(response => response.json()) // преобразуем ответ в JSON  
  .then(data => console.log("Данные:", data))  
  .catch(error => console.error("Ошибка:", error));
```

### Здесь:

- fetch возвращает промис.
- response.json() тоже возвращает промис, поэтому нужен второй .then().

# Результат:

```
[Running] node "c:\Users\user\Documents\JScode\test1\test.js"
```

```
Данные: {  
  userId: 1,  
  id: 1,  
  title: 'sunt aut facere repellat provident occaecati excepturi optio reprehenderit',  
  body: 'quia et suscipit\n' +  
    'suscipit recusandae consequuntur expedita et cum\n' +  
    'reprehenderit molestiae ut ut quas totam\n' +  
    'nostrum rerum est autem sunt rem eveniet architecto'  
}
```

## Пример с async/await.

Тот же запрос можно записать проще:

```
async function getPost() {  
  try {  
    let response = await fetch("https://jsonplaceholder.typicode.com/posts/1");  
    let data = await response.json();  
    console.log("Данные:", data);  
  } catch (error) {  
    console.error("Ошибка:", error);  
  }  
}  
  
getPost();
```

 Код выглядит как «синхронный», но работает асинхронно.

## **Настройка методов и параметров.**

По умолчанию `fetch` делает запрос методом `GET`, но можно указать другие методы HTTP.

## 👉 Пример POST-запроса:

```
fetch("https://jsonplaceholder.typicode.com/posts", {  
  method: "POST", // отправка данных  
  headers: {  
    "Content-Type": "application/json"  
  },  
  body: JSON.stringify({  
    title: "Пример",  
    body: "Текст сообщения",  
    userId: 1  
  })  
})  
  
.then(response => response.json())  
.then(data => console.log("Создано:", data));
```

## Особенности fetch:

- Всегда возвращает промис.
- Ошибка в `.catch()` возникает только при проблемах сети (например, нет интернета).
- Если сервер ответил с кодом ошибки (404, 500), это не считается ошибкой — нужно проверять `response.ok` и `response.status`.



## 👉 Пример проверки:

```
fetch("https://jsonplaceholder.typicode.com/unknown")
  .then(response => {
    if (!response.ok) {
      throw new Error("Ошибка HTTP: " + response.status);
    }
    return response.json();
  })
  .then(data => console.log(data))
  .catch(err => console.error(err));
```

✖ ▶ GET <https://jsonplaceholder.typicode.com/unknown>  
404 (Not Found)

✖ ▶ Error: Ошибка HTTP: 404  
at [test.js:5:13](#)

## Сравнение с XMLHttpRequest:

XMLHttpRequest	fetch
Старый API, громоздкий код	Современный API, лаконичный
Основан на колбэках	Основан на промисах
Требует ручной обработки данных	Умеет работать с .json(), .text(), .blob()
Поддержка всех браузеров, даже очень старых	Поддержка современных браузеров и Node.js 18+

### Вывод:

- fetch — современный и удобный способ выполнения HTTP-запросов.
- Он основан на промисах и легко сочетается с async/await, делая код понятным и читаемым.

## 4. fetch и промисы.

### Что возвращает fetch?

- Вызов `fetch(url)` сразу возвращает промис.
- Этот промис переходит в состояние `fulfilled` (успешно), когда получен ответ от сервера (даже если это ошибка 404 или 500).
- Результат — объект `Response`, который содержит:
  - `status` (код ответа: 200, 404, 500 и т. д.),
  - `ok` (`true`, если статус в диапазоне 200–299),
  - методы для чтения тела ответа: `.json()`, `.text()`, `.blob()`, `.arrayBuffer()`.

👉 Пример:

```
let promise = fetch("https://jsonplaceholder.typicode.com/posts/1");  
console.log(promise); // Promise { <pending> }
```

## Работа с .then()

Так как fetch возвращает промис, можно использовать .then() для обработки ответа.

👉 Пример:

```
fetch("https://jsonplaceholder.typicode.com/posts/1")  
  .then(response => {  
    console.log("Статус:", response.status); // 200  
    return response.json(); // парсим тело ответа в JSON  
  })  
  .then(data => {  
    console.log("Данные:", data); // объект с постом  
  });
```

## Результат:

```
Promise { <pending> }  
Статус: 200  
Данные: {  
  userId: 1,  
  id: 1,  
  title: 'sunt aut facere repellat provident occaecati excepturi optio reprehenderit',  
  body: 'quia et suscipit\n' +  
    'suscipit recusandae consequuntur expedita et cum\n' +  
    'reprehenderit molestiae ut ut quas totam\n' +  
    'nostrum rerum est autem sunt rem eveniet architecto'  
}
```

## Важный момент:

- Первый `.then()` получает объект `Response`.
- Чтобы получить данные, нужно вызвать метод (`.json()`, `.text()`, и т.д.), который тоже возвращает промис.
- Поэтому используется второй `.then()` для получения результата.

## Работа с `.catch()`

Метод `.catch()` нужен для обработки сетевых ошибок (например, если нет интернета или сервер недоступен).

Ошибки HTTP (404, 500) сами по себе не вызывают `.catch()`, их нужно проверять вручную через `response.ok`.



Пример. Здесь .catch() перехватит ошибку, которую мы явно сгенерировали через throw:


```
fetch("https://jsonplaceholder.typicode.com/unknown") // неправильный URL
  .then(response => {
    if (!response.ok) {
      throw new Error("Ошибка HTTP: " + response.status);
    }
    return response.json();
  })
  .then(data => console.log(data))
  .catch(error => console.error("Ошибка запроса:", error.message));
```

```
[Running] node "c:\Users\user\Documents\JSCode\test1\test.js"
Ошибка запроса: Ошибка HTTP: 404
```

## Последовательная обработка.

Можно строить цепочки `.then()` для поэтапной обработки.

```
fetch("https://jsonplaceholder.typicode.com/posts/1")
  .then(res => res.json())
  .then(post => {
    console.log("Заголовок поста:", post.title);
    return fetch("https://jsonplaceholder.typicode.com/users/" + post.userId);
  })
  .then(res => res.json())
  .then(user => console.log("Автор поста:", user.name))
  .catch(err => console.error("Ошибка:", err));
```

 Здесь мы сначала загружаем пост, потом по его `userId` получаем автора. Всё строится на цепочке промисов.

## Результат:

```
[Running] node "c:\Users\user\Documents\JScode\test1\test.js"
```

```
Заголовок поста: sunt aut facere repellat provident occaecati excepturi optio reprehenderit
```

```
Автор поста: Leanne Graham
```

### Вывод:

- `fetch` возвращает промис с объектом `Response`.
- Чтобы получить данные, нужно вызвать метод (`.json()`, `.text()` и др.), который возвращает ещё один промис.
- Ошибки сети обрабатываются через `.catch()`.
- Ошибки HTTP (404, 500) нужно проверять вручную через `response.ok` и `response.status`.

# 5. Использование `async/await` для HTTP-запросов

Зачем использовать `async/await`?

- Работа с `fetch` через `.then()` и `.catch()` может делать код вложенным и менее читаемым.
- `async/await` позволяет писать асинхронный код в пошаговом стиле, похожем на обычный синхронный.
- Это упрощает логику и особенно полезно, когда нужно сделать несколько запросов подряд.

## Пример:

```
async function getPost() {  
  try {  
    let response = await fetch("https://jsonplaceholder.typicode.com/posts/1");  
    let data = await response.json(); // ждём, пока ответ превратится в JSON  
    console.log("Пост:", data);  
  } catch (error) {  
    console.error("Ошибка сети:", error);  
  }  
}  
  
getPost();
```

 Здесь:

- `await fetch(...)` ждёт завершения запроса.
- `await response.json()` ждёт преобразования ответа в JSON.
- Ошибки (например, если нет интернета) ловятся в `try/catch`.

## Результат:

```
[Running] node "c:\Users\user\Documents\JSCode\test1\test.js"
```

```
Пост: {  
  userId: 1,  
  id: 1,  
  title: 'sunt aut facere repellat provident occaecati excepturi optio reprehenderit',  
  body: 'quia et suscipit\n' +  
    'suscipit recusandae consequuntur expedita et cum\n' +  
    'reprehenderit molestiae ut ut quas totam\n' +  
    'nostrum rerum est autem sunt rem eveniet architecto'  
}
```

## **Проверка успешности ответа.**

Важно помнить: `fetch` не выбрасывает ошибку при кодах 404 или 500.

Поэтому нужно проверять `response.ok`.

📌 Здесь при 404 запрос дойдёт до throw, и управление перейдёт в catch.

```
async function getData() {  
  try {  
    let response = await fetch("https://jsonplaceholder.typicode.com/unknown");  
    if (!response.ok) {  
      throw new Error("Ошибка HTTP: " + response.status);  
    }  
    let data = await response.json();  
    console.log("Данные:", data);  
  } catch (error) {  
    console.error("Запрос не удался:", error.message);  
  }  
}  
getData();
```



# Последовательные запросы.

Если нужно выполнить несколько запросов один за другим:

```
async function getPostAndUser() {  
  try {  
    let postRes = await fetch("https://jsonplaceholder.typicode.com/posts/1");  
    let post = await postRes.json();  
    let userRes = await fetch("https://jsonplaceholder.typicode.com/users/"  
      + post.userId);  
    let user = await userRes.json();  
    console.log("Пост:", post.title);  
    console.log("Автор:", user.name);  
  } catch (e) {  
    console.error("Ошибка:", e);  
  }  
}  
getPostAndUser();
```

# Параллельные запросы.

Если запросы независимы, их лучше выполнять параллельно через `Promise.all`. Оба запроса выполняются одновременно, что экономит время.

```
async function getParallel() {  
  try {  
    let [postsRes, usersRes] = await Promise.all([  
      fetch("https://jsonplaceholder.typicode.com/posts"),  
      fetch("https://jsonplaceholder.typicode.com/users")  
    ]);  
    let posts = await postsRes.json();  
    let users = await usersRes.json();  
    console.log("Всего постов:", posts.length);  
    console.log("Всего пользователей:", users.length);  
  } catch (e) {  
    console.error("Ошибка:", e);  
  }  
}
```

`getParallel();`

## Вывод:

- `async/await` делает работу с `fetch` более простой и наглядной.
- Можно использовать `try/catch` для перехвата ошибок.
- Для последовательных операций код выглядит линейным.
- Для параллельных запросов удобно сочетать `await` с `Promise.all`.

## 6. Обработка ошибок: сетевые ошибки и статус ответа.

При работе с `fetch` и HTTP-запросами можно столкнуться с двумя основными типами ошибок:

- Сетевые ошибки — когда запрос вообще не дошёл до сервера (например, нет интернета, неверный домен).
- Ошибки статуса ответа — сервер ответил, но с кодом, который указывает на проблему (404 — «не найдено», 500 — «внутренняя ошибка»).

## Сетевые ошибки.

Если сеть недоступна или сервер «не отвечает», промис `fetch` переходит в `reject`, и ошибка ловится в `.catch()` или `try/catch`.

## 👉 Пример:

```
async function networkErrorDemo() {  
  try {  
    let response = await fetch("https://wrong-domain.example.com/data");  
    let data = await response.json();  
    console.log(data);  
  } catch (err) {  
    console.error("Сетевая ошибка:", err.message);  
  }  
}  
networkErrorDemo();
```

```
[Running] node "c:\Users\user\Documents\JScode\test1\test.js"  
Сетевая ошибка: fetch failed
```

📌 Здесь catch перехватит ошибку (например, «Failed to fetch»).

## Ошибки статуса ответа.

Важно: `fetch` не выбрасывает ошибку, если сервер вернул 404 или 500.

Вместо этого нужно вручную проверить свойства ответа:

- `response.ok` → `true`, если статус в диапазоне 200–299.
- `response.status` → сам код ответа.

## Пример:

```
async function statusErrorDemo() {  
  let response = await fetch("https://jsonplaceholder.typicode.com/unknown");  
  if (!response.ok) {  
    console.error("Ошибка HTTP:", response.status); // 404  
    return;  
  }  
  let data = await response.json();  
  console.log(data);  
}  
statusErrorDemo();
```

 В этом примере сервер вернёт 404, поэтому данные не будут обработаны.



## Распространённые коды ответов:


- 200 — OK (успех, данные получены).
- 201 — Created (новый ресурс создан, например при POST-запросе).
- 400 — Bad Request (ошибка клиента, неправильный запрос).
- 401 — Unauthorized (нужна авторизация).
- 403 — Forbidden (нет прав доступа).
- 404 — Not Found (ресурс не найден).
- 500 — Internal Server Error (ошибка на стороне сервера).

Все коды: <https://http.cat/>

## Комбинированный подход.

Можно объединить проверку сети и статуса в одном блоке try/catch:

```
async function safeFetch() {  
  try {  
    let response = await fetch("https://jsonplaceholder.typicode.com/unknown");  
    if (!response.ok) {  
      throw new Error("Ошибка HTTP: " + response.status);  
    }  
    let data = await response.json();  
    console.log("Данные:", data);  
  } catch (error) {  
    console.error("Запрос не удался:", error.message);  
  }  
}  
safeFetch();
```

 Здесь:

- Если нет интернета → попадём в catch.
- Если сервер вернул 404 → сработает throw, и тоже попадём в catch.

## Вывод:

- `fetch` может завершиться ошибкой из-за проблем с сетью.
- Ошибки HTTP (404, 500) не считаются сетевыми — их нужно проверять через `response.ok`.
- Хороший стиль: всегда оборачивать запросы в `try/catch` и проверять `response.status`.

# Практические примеры: работа с публичными API.

При работе с fetch и HTTP-запросами можно столкнуться с двумя основными типами ошибок:

- Сетевые ошибки — когда запрос вообще не дошёл до сервера (например, нет интернета, неверный домен).
- Ошибки статуса ответа — сервер ответил, но с кодом, который указывает на проблему (404 — «не найдено», 500 — «внутренняя ошибка»).

Например:

1. Курсы валют. ExchangeRate API: [exchangerate-api.com](https://exchangerate-api.com)
2. Криптовалюты. CoinGecko API: [coingecko.com/en/api](https://coingecko.com/en/api)
3. Погода. OpenWeatherMap: [openweathermap.org/api](https://openweathermap.org/api)
4. Случайные факты. Random Facts API: [uselessfacts.jsph.pl](https://uselessfacts.jsph.pl)
5. Цитаты. Quotable API: [quotable.io](https://quotable.io)
6. Генерация случайных данных. [randomuser.me](https://randomuser.me)
7. Данные о животных.  
Dog API: [dog.ceo/dog-api](https://dog.ceo/dog-api)  
Cat API: [thecatapi.com](https://thecatapi.com).
8. Рецепты. Recipe API: [spoonacular.com/food-api](https://spoonacular.com/food-api)
9. Факты о стране. REST Countries API: [restcountries.com](https://restcountries.com)

# Курсы валют — ExchangeRate API

```
async function getRates() {  
  let res = await fetch("https://open.er-api.com/v6/latest/USD");  
  let data = await res.json();  
  console.log("Курс USD к EUR:", data.rates.EUR);  
  console.log("Курс USD к JPY:", data.rates.JPY);  
}  
  
getRates();
```

```
[Running] node "c:\Users\user\Documents\JScode\test1\test.js"  
Курс USD к EUR: 0.853717  
Курс USD к JPY: 147.548586
```

## Данные о стране — REST Countries API

```
async function getCountry() {  
  let res = await fetch("https://restcountries.com/v3.1/name/qazaqstan");  
  let data = await res.json();  
  console.log("Страна:", data[0].name.common);  
  console.log("Столица:", data[0].capital[0]);  
  console.log("Регион:", data[0].region);  
}  
  
getCountry();
```

Страна: Kazakhstan

Столица: Astana

Регион: Asia

# Контрольные вопросы:

- Что такое HTTP-запросы и как они связаны с асинхронностью в JS?
- Чем отличается XMLHttpRequest от fetch?
- Что возвращает метод fetch?
- Почему fetch удобнее использовать с промисами?
- Как обрабатывать ошибки при запросах?
- Чем отличается ошибка сети от статуса ответа сервера?
- Как использовать async/await для написания более читаемого кода запросов?
- В каких случаях всё ещё может использоваться XMLHttpRequest?



# Домашнее задание:

1. <https://ru.hexlet.io/courses/js-asynchronous-programming>

## 15 HTTP-запросы

Учимся пользоваться асинхронностью в прикладных задачах (на примере HTTP-клиента axios)

## 16 Дополнительные материалы

2. Повторить материал лекции.

# **Материалы лекций:**

<https://github.com/ShViktor72/Education2025>

# **Обратная связь:**

[colledge20education23@gmail.com](mailto:colledge20education23@gmail.com)