

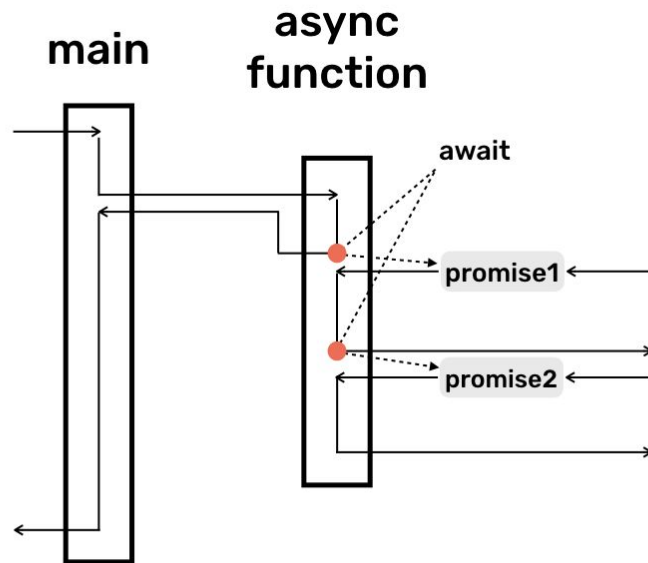


# Тема занятия:

## Async/Await

### JS Fetch

# async/await



**Async/await** — это специальный синтаксис, который предназначен для более простого и удобного написания асинхронного кода. Синтаксис «async/await» упрощает работу с промисами.

Появился он в языке, начиная с ES2017 (**ES8**)

```
async() {  
    await()  
}
```

**JS**

# Ключевое слово `await`

Ключевое слово **`await`** заставит интерпретатор JavaScript ждать до тех пор, пока промис справа от **`await`** не выполнится. После чего оно вернёт его результат, и выполнение кода продолжится.

```
1  async function f() {  
2  
3      let promise = new Promise((resolve, reject) => {  
4          setTimeout(() => resolve("готово!"), 1000)  
5      });  
6  
7      let result = await promise; // будет ждать, пока промис не выполнится (*)  
8  
9      alert(result); // "готово!"  
10 }  
11  
12 f();
```

**Примечание:** если мы попробуем использовать `await` внутри функции, объявленной без `async`, получим синтаксическую ошибку

# Обработка ошибок

На практике промис может завершиться с ошибкой не сразу, а через некоторое время. В этом случае будет задержка, а затем `await` выбросит исключение.

Такие ошибки можно ловить, используя `try..catch`

```
1  async function f() {  
2  
3    try {  
4      let response = await fetch('/no-user-here');  
5      let user = await response.json();  
6    } catch(err) {  
7      // перехватит любую ошибку в блоке try: и в fetch, и в response.json  
8      alert(err);  
9    }  
10 }  
11  
12 f();
```

# Обработка ошибок

При работе с `async/await`, `.then` используется нечасто, так как `await` автоматически ожидает завершения выполнения промиса. В этом случае обычно гораздо удобнее перехватывать ошибки, используя `try..catch`.

Но на верхнем уровне вложенности (вне `async`-функций) `await` использовать нельзя, поэтому `.then/catch` для обработки финального результата или ошибок – обычная практика.

```
1  async function f() {  
2    let response = await fetch('http://no-such-url');  
3  }  
4  
5  // f() вернёт промис в состоянии rejected  
6  f().catch(alert); // TypeError: failed to fetch // (*)
```

# async/await отлично работает с Promise.all

Когда необходимо подождать несколько промисов одновременно, можно обернуть их в **Promise.all**, и затем **await**:

```
async function f() {  
  let results = await Promise.all([  
    fetch(url1),  
    fetch(url2),  
    ...  
  ]);  
}
```

Promise 1

Promise 2

Promise 3

# Итог:

Ключевое слово **async** перед объявлением функции:

1. Обязывает её всегда возвращать промис.
2. Позволяет использовать **await** в теле этой функции.

Ключевое слово **await** перед промисом заставит JavaScript дожждаться его выполнения, после чего:

1. Если промис завершается с ошибкой, будет сгенерировано исключение, как если бы на этом месте находилось **throw**.
2. Иначе вернётся результат промиса.

Вместе они предоставляют отличный каркас для написания асинхронного кода. Такой код легко и писать, и читать.





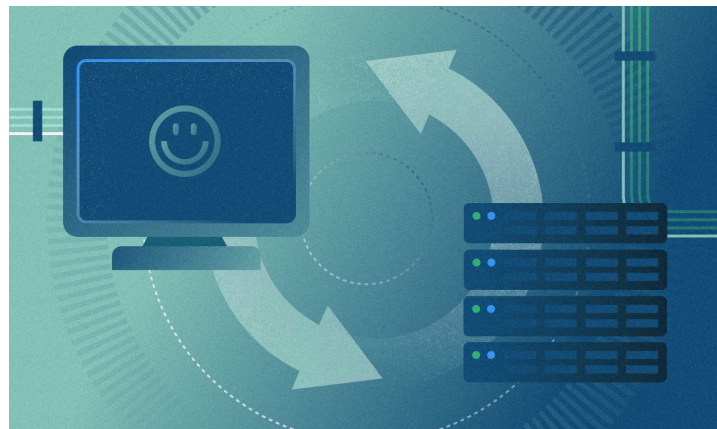
The image features decorative geometric patterns in the corners. The top-left and bottom-right corners contain overlapping blue and dark blue shapes with gold outlines and a pattern of small blue dots. The text "JS Fetch" is centered in the white space between these patterns.

# JS Fetch

JavaScript может отправлять сетевые запросы на сервер и подгружать новую информацию по мере необходимости.

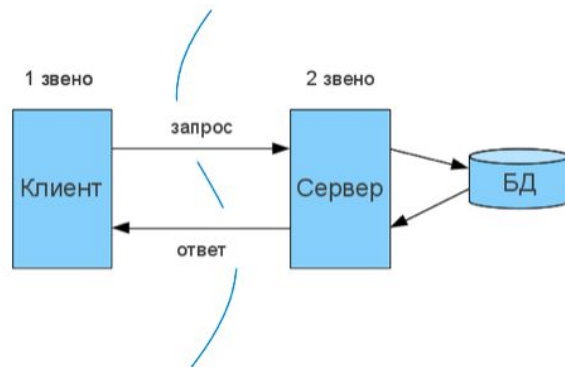
Например, мы можем использовать сетевой запрос, чтобы:

- Отправить заказ,
- Загрузить информацию о пользователе,
- Запросить последние обновления с сервера,
- ...и т.п.

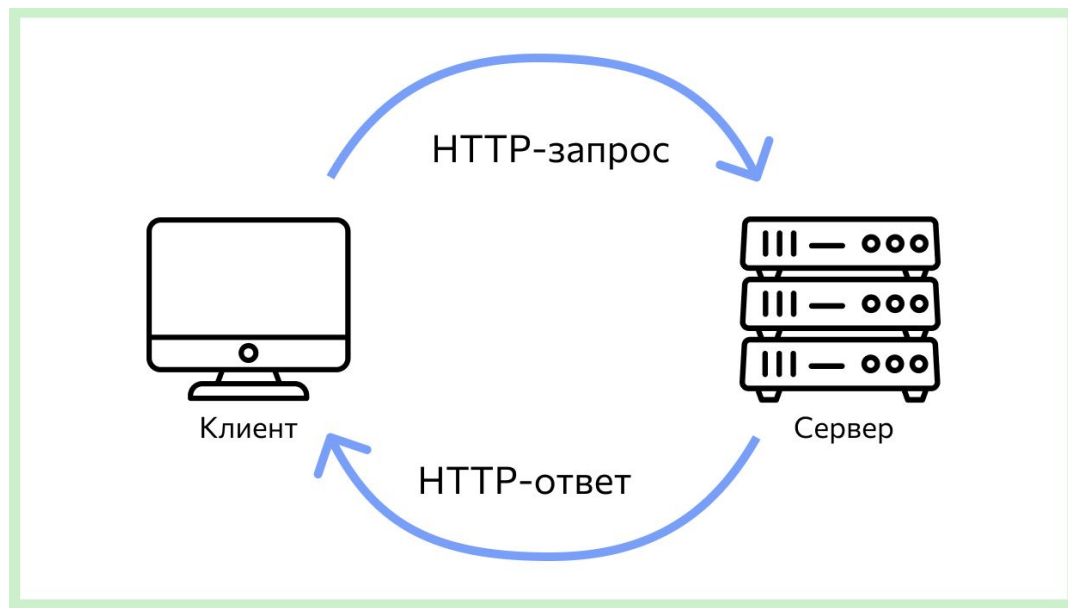


В клиент-серверной архитектуре используется три компонента:

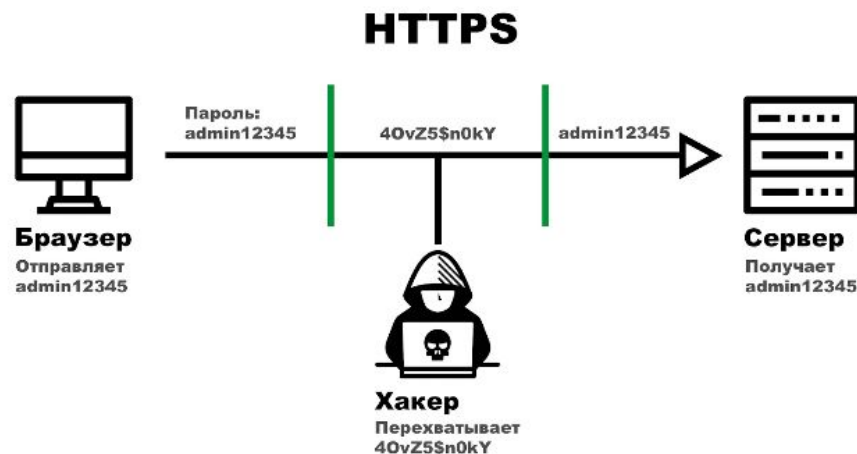
- **Клиент** — программа, которую мы используем в интернете. Чаще всего это браузер, но может быть и другая отдельная программа
- **Сервер** — компьютер, на котором хранится сайт или приложение. Когда мы заходим на сайт магазина, мы обращаемся к серверу, на котором находится сайт
- **База данных** — программа, в которой хранятся все данные приложения.



**HTTP** – это протокол передачи информации в интернете, который расшифровывается как «протокол передачи гипертекста» (HyperText Transfer Protocol).



**HTTPS** — это расширение для протокола HTTP, которое делает его безопасным. Дело в том, что данные передаются по HTTP в открытом виде. HTTPS решает эту проблему, добавляя в изначальный протокол возможность шифрования данных.

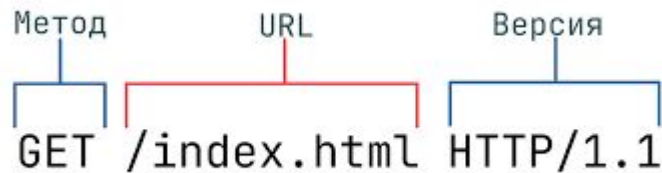


## HTTP-запрос состоит из трех элементов:

- **стартовой строки**, которая задает параметры запроса или ответа,
- **заголовка**, который описывает сведения о передаче и другую служебную информацию.
- **тело** (его не всегда можно встретить в структуре). Обычно в нем как раз лежат передаваемые данные. От заголовка тело отделяется пустой строкой.



# Стартовая строка



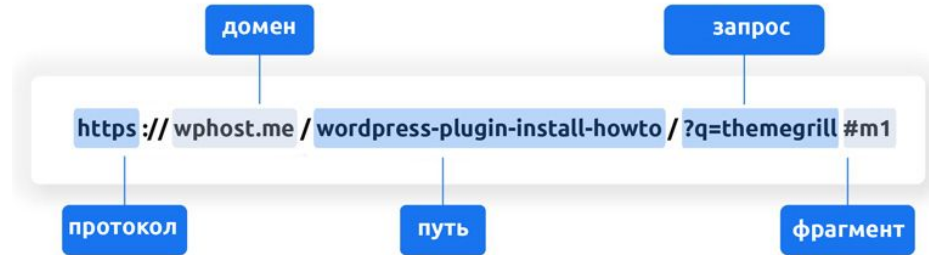
**Метод** – описывает, какое именно действие нужно совершить со страницей.

Самые популярные:

- GET (получение данных)
- POST (отправка данных)
- PUT (отправка данных)
- DELETE (удаление)



**URL** (Uniform Resource Locator) – единообразный идентификатор ресурса, идентифицирует ресурс и определяет его точное местоположение. Именно с помощью URL записаны ссылки в интернете.



**Версия** определяет, в соответствии с какой версией стандарта HTTP составлен запрос. Указывается как два числа, разделённых точкой (например 1.1).



# headers (заголовки)

Заголовки HTTP позволяют клиенту и серверу отправлять дополнительную информацию с HTTP запросом или ответом

POST / HTTP/1.1

Host: example.com

User-Agent: Mozilla/5.0 (X11;...) Firefox/91.0

Accept: text/html, application/json

Accept-Language: ru-RU

Accept-Encoding: gzip, deflate

Connection: keep-alive

Upgrade-Insecure-Requests: 1

Content-Type: multipart/form-data; boundary=b4e4fbd93540

Content-Length: 345

Заголовки  
запроса

Заголовки общего  
назначения

Заголовки  
представления

# body (тело)

Тело сообщения опционально, оно содержит данные, связанные с запросом, либо документ (например HTML-страницу), передаваемый в ответе. Некоторые виды запросов могут отправлять данные на сервер в теле запроса

```
POST /?id=1 HTTP/1.1
```

## Request line

```
Host: www.swingvy.com
Content-Type: application/json; charset=utf-8
User-Agent: Mozilla/5.0 (Macintosh; Intel Mac OS X 10.12; rv:53.0)
Gecko/20100101 Firefox/53.0
Connection: close
Content-Length: 136
```

## Header

```
{
  "status": "ok",
  "extended": true,
  "results": [
    {"value": 0, "type": "int64"},
    {"value": 1.0e+3, "type": "decimal"}
  ]
}
```

## Body message

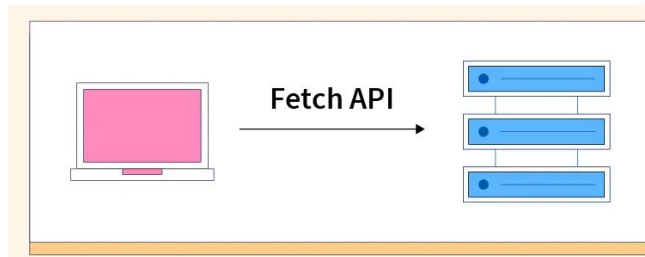
Категория	Описание
200 OK	Возвращается в случае успешной обработки запроса, при этом тело ответа обычно содержит запрошенный ресурс.
302 Found	Перенаправляет клиента на другой URL. Например, данный код может прийти, если клиент успешно прошел процедуру аутентификации и теперь может перейти на страницу своей учетной записи.
400 Bad Request	Данный код можно увидеть, если запрос был сформирован с ошибками. Например, в нем отсутствовали символы завершения строки.
403 Forbidden	Означает, что клиент не обладает достаточными правами доступа к запрошенному ресурсу. Также данный код можно встретить, если сервер обнаружил вредоносные данные, отправленные клиентом в запросе.
404 Not Found	Каждый из нас, так или иначе, сталкивался с этим кодом ошибки. Данный код можно увидеть, если запросить у сервера ресурс, которого не существует на сервере.
500 Internal Error	Данный код возвращается сервером, когда он не может по определенным причинам обработать запрос.

**Метод `fetch()`** — современный и очень мощный способ для создания сетевых запросов и получения информации с сервера

```
1 let promise = fetch(url, [options])
```

**url** — URL для отправки запроса.

**options** — дополнительные параметры: метод, заголовки и так далее.



**fetch()** запускает запрос и возвращает promise.

Когда запрос удовлетворяется, promise разрешается(**resolved**) объектом ответа (Response object)

Если ответ не получается получить из-за, например, неполадок сети, promise отклоняется (promise is **rejected**).



**async/await** синтаксис очень помогает при работе с `fetch()`

Для примера сделаем запрос, чтобы получить информацию о фильмах:

```
async function fetchMovies() {  
  const response = await fetch('/movies');  
  // waits until the request completes...  
  console.log(response);  
}
```



Объект **Response** имеет некоторые свойства, с помощью которых можно обращаться к некоторым данным ответа

**Response.status** – читает ответ и возвращает как обычный текст,

**Response.statusText** – возвращает ответ как объект

**Response.ok** – булевское значение, которое указывает, выполнен ли запрос успешно или нет

**Response.headers** – объект, который описывает заголовок ответа.



Существует метод, который помогает извлечь из объекта **Response** информацию ответа в JSON формате:

**Response.json()**

Этот метод тоже возвращает promise, поэтому его необходимо дожидаться при помощи await:

```
async function f() {  
  let url = 'https://api.github.com/repos/javascript-tutorial/commits';  
  let response = await fetch(url);  
  let commits = await response.json(); // читаем ответ в формате JSON  
  alert(commits[0].author);  
}
```



# Работа с ошибками



Промис завершается с ошибкой, если `fetch` не смог выполнить HTTP-запрос, например при ошибке сети или если нет такого сайта. HTTP-статусы 404 и 500 (и другие) не являются ошибкой.

```
1 let response = await fetch(url);
2
3 if (response.ok) { // если HTTP-статус в диапазоне 200-299
4   // получаем тело ответа (см. про этот метод ниже)
5   let json = await response.json();
6 } else {
7   alert("Ошибка HTTP: " + response.status);
8 }
```

# Метод GET

Метод GET используется для получения данных с сервера.

```
const response = await fetch('https://api.example.com/data', {  
  method: 'GET',  
  headers: {  
    'Content-Type': 'application/json',  
    // Дополнительные заголовки, если необходимо  
  },  
  // Дополнительные параметры запроса, например, параметры URL  
  // Также можно использовать URLSearchParams для удобной работы с параметрами  
});  
  
const data = await response.json();  
console.log('GET data:', data);
```

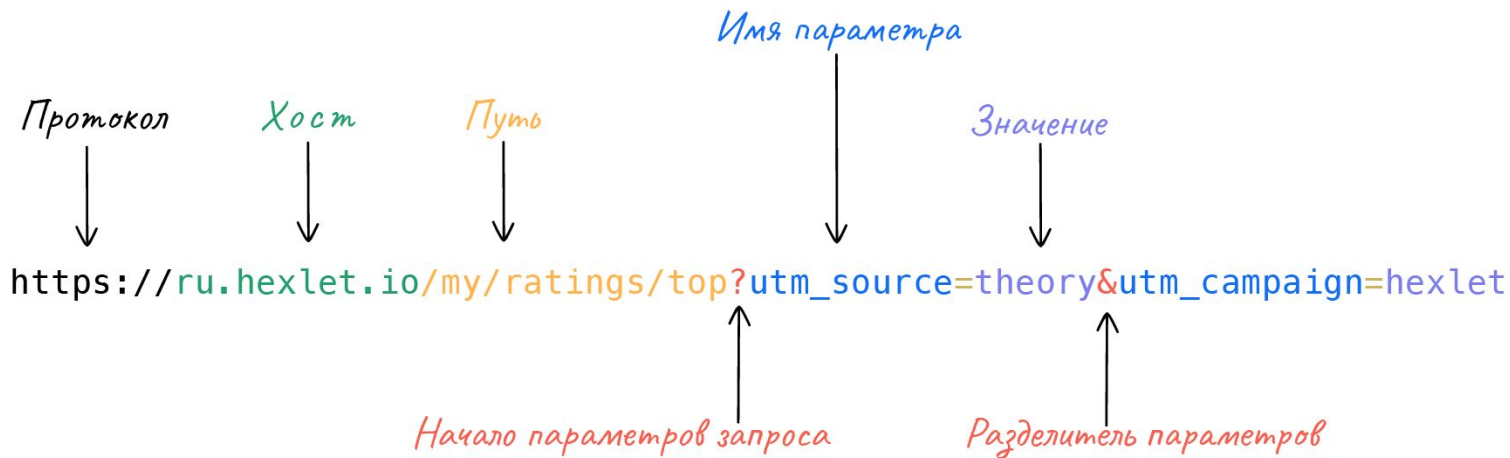
Для метода GET можно не указывать options

```
const response = await fetch('https://api.example.com/data');
```

# Метод GET

GET-запросы, по определению, не предназначены для отправки данных на сервер.

Однако, иногда может потребоваться отправить данные в URL в виде параметров запроса.



# Метод POST

Метод POST отправляет данные на сервер и создает новый ресурс

```
const response = await fetch('https://api.example.com/data', {
  method: 'POST',
  headers: {
    'Content-Type': 'application/json',
    // Дополнительные заголовки, если необходимо
  },
  body: JSON.stringify({
    key1: 'value1',
    key2: 'value2',
    // Данные для отправки на сервер
  }),
});

const data = await response.json();
console.log('POST data:', data);
```

# Метод PUT

Метод PUT чаще всего используется для обновления существующего ресурса. Для этого необходим URI ресурса и новая его версия.

```
const response = await fetch('https://api.example.com/data/123', {
  method: 'PUT',
  headers: {
    'Content-Type': 'application/json',
    // Дополнительные заголовки, если необходимо
  },
  body: JSON.stringify({
    key1: 'updatedValue1',
    key2: 'updatedValue2',
    // Обновленные данные для отправки на сервер
  }),
});

const data = await response.json();
console.log('PUT data:', data);
```

# Метод DELETE

Метод DELETE используется для удаления ресурса, который указывается с помощью его URI.

```
const response = await fetch('https://api.example.com/data/123', {  
  method: 'DELETE',  
  headers: {  
    'Content-Type': 'application/json',  
    // Дополнительные заголовки, если необходимо  
  },  
});  
  
const data = await response.json();  
console.log('DELETE data:', data);
```