JS: async/await, fetch

НАШИ ПРАВИЛА

Включенная камера

Вопросы по поднятой руке

Не перебиваем друг друга

Все вопросы, не связанные с тематикой курса (орг-вопросы и т. д.), должны быть направлены куратору

Подготовьте свое рабочее окружение для возможной демонстрации экрана (закройте лишние соцсети и прочие приложения)

Повторим ;)

Что такое promise?

Что нужно передать в promise при его создании?

Назовите 3 состояния promise

ЦЕЛЬ

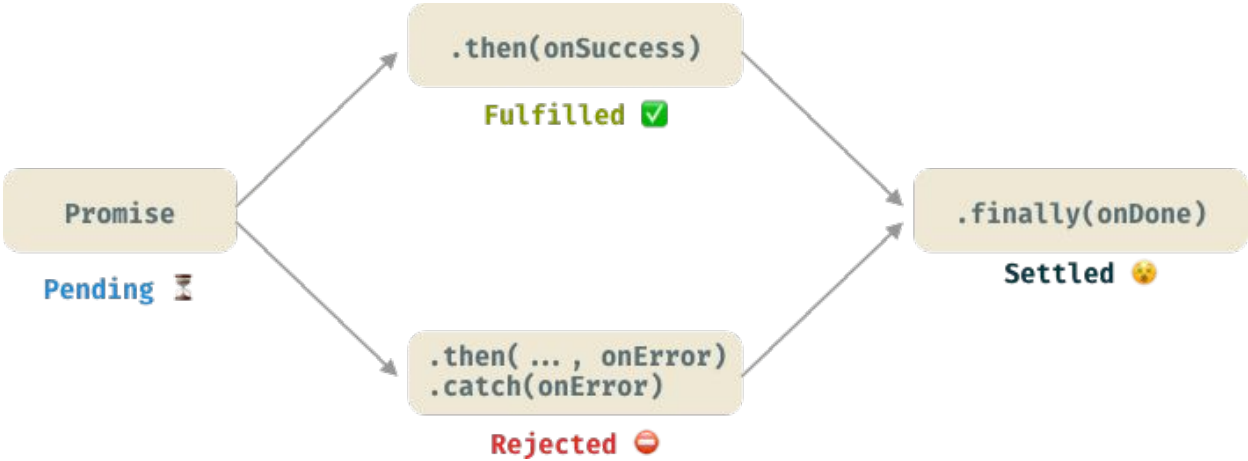
Познакомиться с конструкцией async/await и клиент-серверной архитектурой

ПЛАН ЗАНЯТИЯ

- Async await
- Клиент-серверная архитектура
- fetch

Существует три метода, которые позволяют работать с результатом выполнения вычисления внутри промиса:

- then()
- catch()
- finally()



Метод then()

Первый аргумент метода .then – функция, которая выполняется, когда промис переходит в состояние «выполнен успешно», и получает результат.

Второй аргумент .then – функция, которая выполняется, когда промис переходит в состояние «выполнен с ошибкой», и получает ошибку.

```
promise.then(
  function(result) { /* обработает успешное выполнение */ },
  function(error) { /* обработает ошибку */ }
);
```

Метод catch()

Если мы хотели бы только обработать ошибку, то можно использовать null в качестве первого аргумента: .then(null, errorHandlingFunction). Или можно воспользоваться методом .catch(errorHandlingFunction), который сделает то же самое:

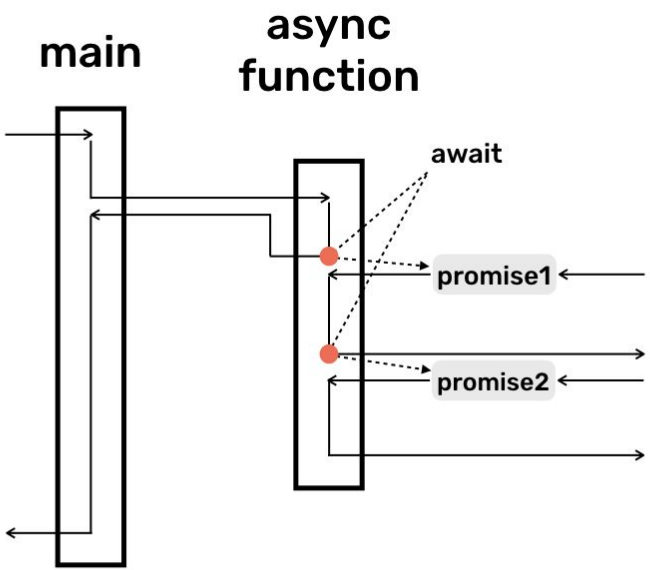
```
let promise = new Promise((resolve, reject) => {
  setTimeout(() => reject(new Error("Ошибка!")), 1000);
});

// .catch(f) это то же самое, что promise.then(null, f)
promise.catch(alert); // выведет "Error: Ошибка!" спустя одну секунду
```

Метод finally()

Вызов .finally(f) похож на .then(f, f), в том смысле, что f выполнится в любом случае, когда промис завершится: успешно или с ошибкой.

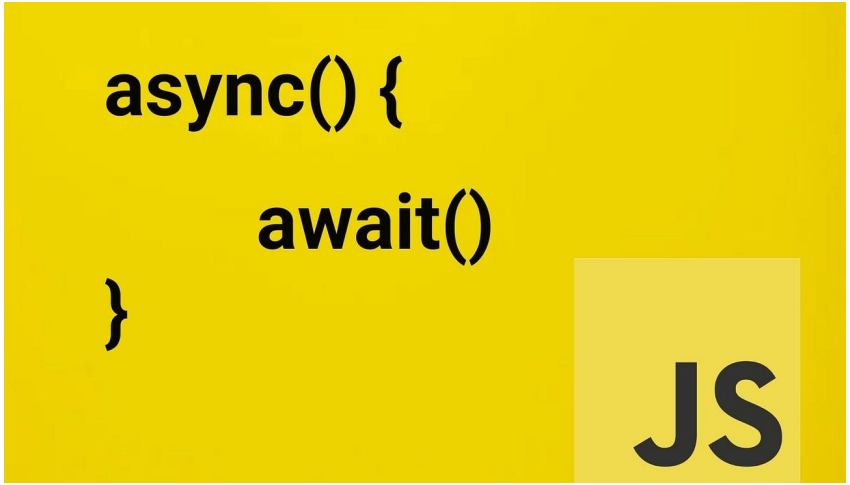
```
new Promise((resolve, reject) => {
  /* сделать что-то, что займёт время, и после вызвать resolve или может reject */
})
  // выполнится, когда промис завершится, независимо от того, успешно или нет
  .finally(() => остановить индикатор загрузки)
  // таким образом, индикатор загрузки всегда останавливается, прежде чем мы продолжим
  .then(result => показать результат, err => показать ошибку)
```

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)



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.

```
async function f() {

  let promise = new Promise((resolve, reject) => {
    setTimeout(() => resolve("готово!"), 1000)
  });

  let result = await promise; // будет ждать, пока промис не выполнится (*)

  alert(result); // "готово!"
}

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

```
async function f() {

  try {
    let response = await fetch('/no-user-here');
    let user = await response.json();
  } catch(err) {
    // перехватит любую ошибку в блоке try: и в fetch, и в response.json
    alert(err);
  }
}

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.

```
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 // (*)
```

Итог:

Ключевое слово async перед объявлением функции:

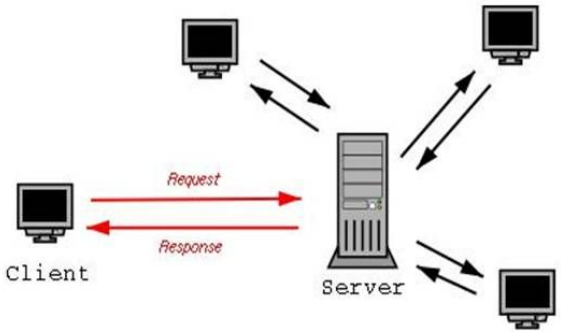
1. Обязывает её всегда возвращать промис.
2. Позволяет использовать await в теле этой функции.

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

1. Если промис завершается с ошибкой, будет сгенерировано исключение
2. Иначе вернётся результат промиса.

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

Клиент-серверная архитектура



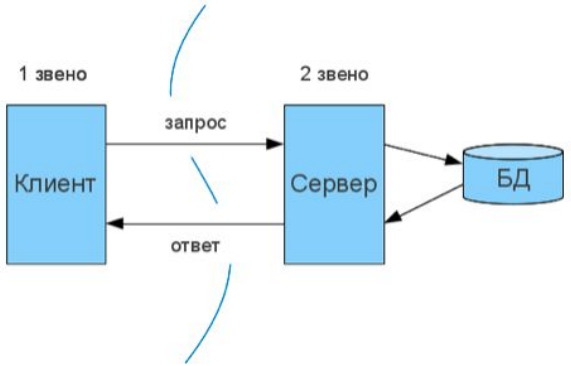
JavaScript может отправлять сетевые запросы на сервер и подгружать новую информацию по мере необходимости.

Например, мы можем использовать сетевой запрос, чтобы:

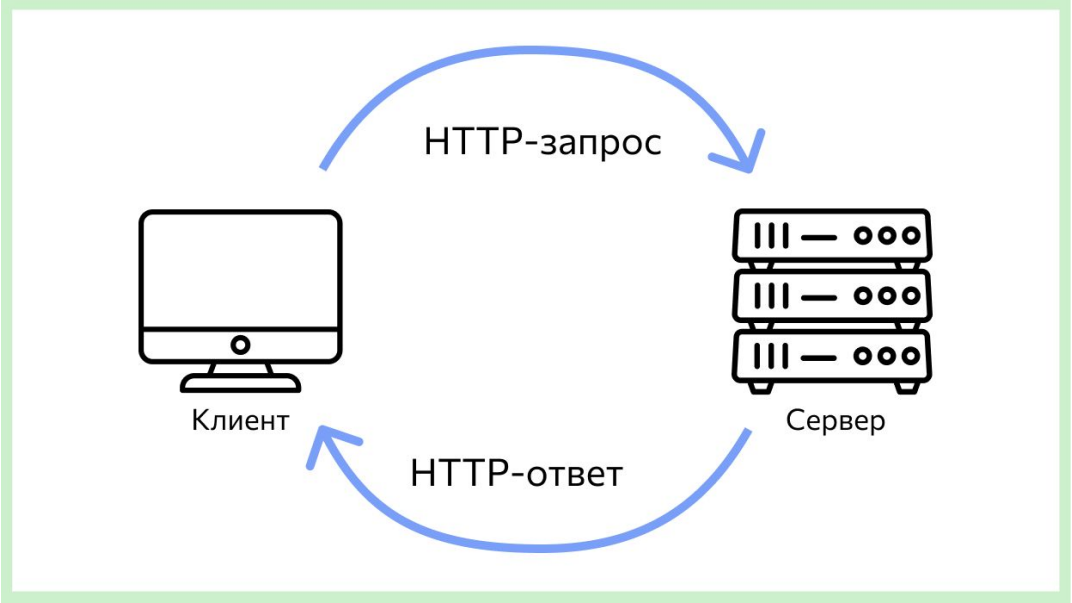
- Отправить заказ,
- Загрузить информацию о пользователе,
- Запросить последние обновления с сервера,
- …и т.п.

В клиент-серверной архитектуре используется три компонента:

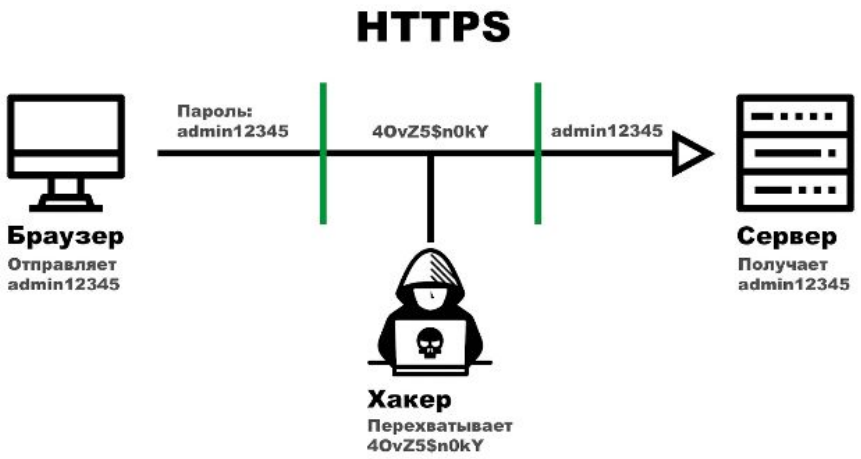
- **Клиент** — программа, которую мы используем в интернете. Чаще всего это браузер, но может быть и другая отдельная программа
- **Сервер** — компьютер, на котором хранится сайт или приложение. Когда мы заходим на сайт магазина, мы обращаемся к серверу, на котором находится сайт
- **База данных** —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

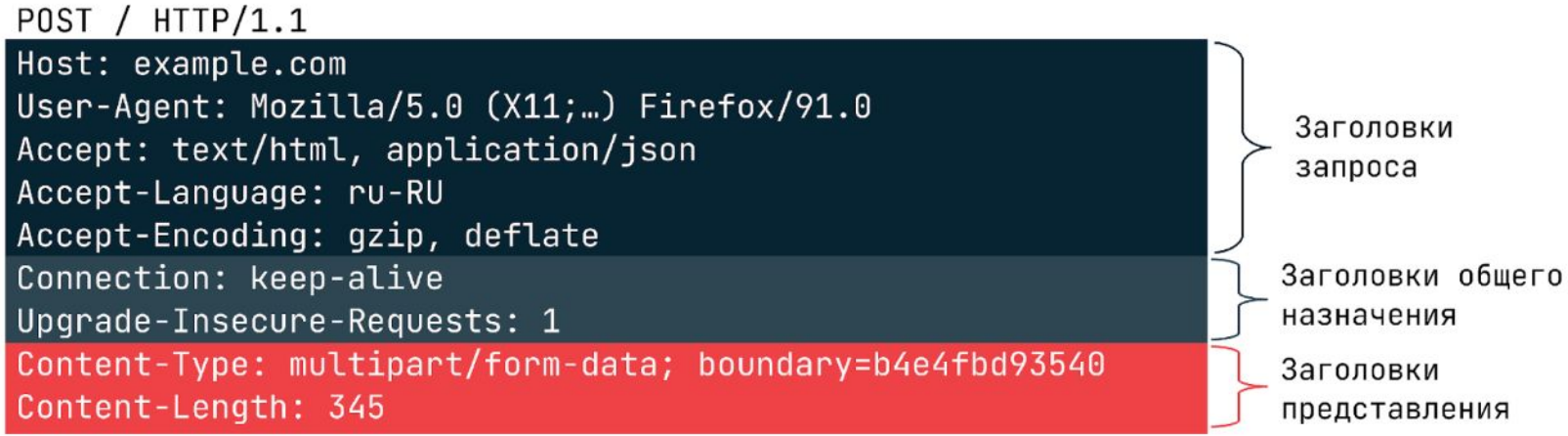
Метод – описывает, какое именно действие нужно совершить со страницей.

Самые популярные:

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

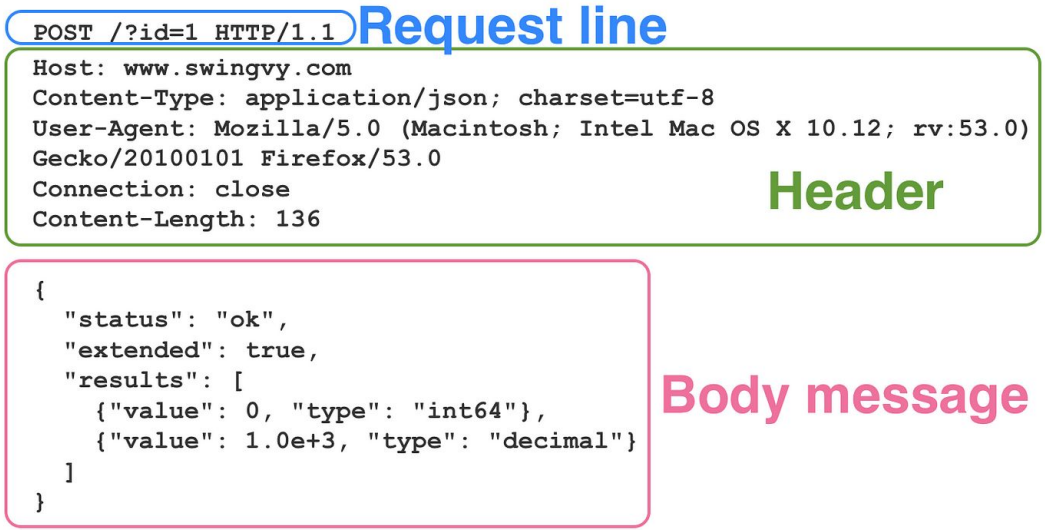
headers (заголовки)

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



body (тело)

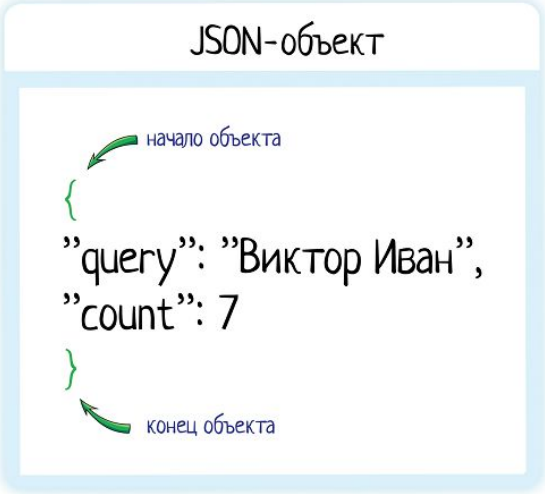
Тело сообщения опционально, оно содержит данные, связанные с запросом, либо документ (например HTML-страницу), передаваемый в ответе. Некоторые виды запросов могут отправлять данные на сервер в теле запроса



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.


{JSON}
JavaScript Object Notation

JSON - текстовый формат данных, следующий за синтаксисом объекта JavaScript, который был популяризирован Дугласом Крокфордом. Несмотря на то, что он очень похож на буквенный синтаксис объекта JavaScript, его можно использовать независимо от JavaScript, и многие среды программирования имеют возможность читать (анализировать) и генерировать JSON.



JSON представляет собой коллекцию пар ключ-значение. Ключи и значения разделяются двоеточием, а пары ключ-значение - запятой. Объект начинается с { и заканчивается }.

```
{
  "orderID": 12345,
  "shopperName": "Ivan Ivanov",
  "shopperEmail": "ivanov@example.com",
  "contents": [
    {
      "productID": 34,
      "productName": "Super product",
      "quantity": 1
    },
    {
      "productID": 56,
      "productName": "Wonderful product",
      "quantity": 3
    }
  ],
  "orderCompleted": true
}
```

Примечания

- JSON - это чисто формат данных - он содержит только свойства, без методов.
- JSON требует двойных кавычек, которые будут использоваться вокруг строк и имён свойств. Одиночные кавычки недействительны.
- Даже одна неуместная запятая или двоеточие могут привести к сбою JSON-файла и не работать.

Методы работы с JSON

1. Преобразует JavaScript объект в JSON-строку - **`JSON.stringify(obj)`**

```
const obj = { name: "John", age: 30 };
const jsonString = JSON.stringify(obj);
```

1. Преобразует JSON-строку в JavaScript объект - **`JSON.parse(jsonString)`**

```
const jsonString = '{"name":"John","age":30}';
const obj = JSON.parse(jsonString);
```

- Простой и легко читаемый синтаксис, который понятен и компьютеру, и человеку.
- Лёгкость и компактность. JSON-файлы весят меньше, чем файлы других форматов, и загружаются быстрее.
- Отсутствие зависимости от конкретного языка программирования.
- Универсальность в типах данных. JSON можно использовать для хранения массивов, упорядоченных списков и коллекций пар «ключ — значение».
- Широкая поддержка в браузерах. Все современные браузеры поддерживают работу с JSON.
- Самостоятельное документирование.

XML

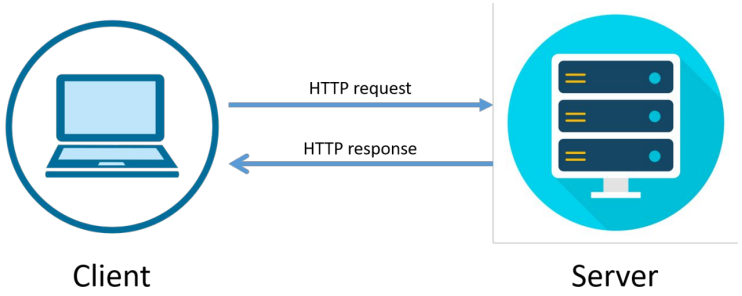
```xml
<?xml version="1.0" encoding="UTF-8" ?>
<person>
 <name>Иван</name>
 <age>37</age>
 <mother>
  <name>Ольга</name>
  <age>58</age>
 </mother>
 <children>
  <child>Маша</child>
  <child>Игорь</child>
  <child>Таня</child>
 </children>
 <married>true</married>
 <dog null="true" />
</person>
```

VS

JSON

```json
{
 "person":{
  "name":"Иван",
  "age":37,
  "mother":{
   "name":"Ольга",
   "age":58
  },
  "children":[
   "Маша",
   "Игорь",
   "Таня"
  ],
  "married":true,
  "dog":null
 }
```

Fetch

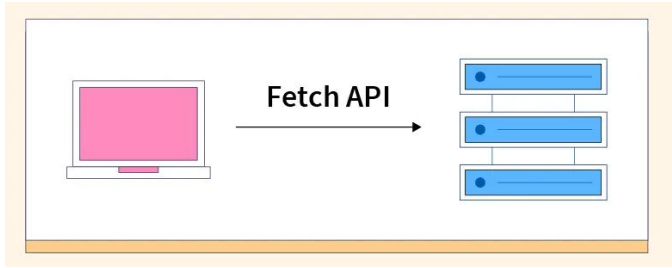


Метод fetch() — современный и очень мощный способ для создания сетевых запросов и получения информации с сервера

```
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– возвращает статус код ответа

Response.statusText – возвращает текст для статус кода

Response.ok – булевское значение, которое указывает, выполнился 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.

```
let response = await fetch(url);

if (response.ok) { // если HTTP-статус в диапазоне 200-299
  // получаем тело ответа (см. про этот метод ниже)
  let json = await response.json();
} else {
  alert("Ошибка HTTP: " + response.status);
}
```

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);
```

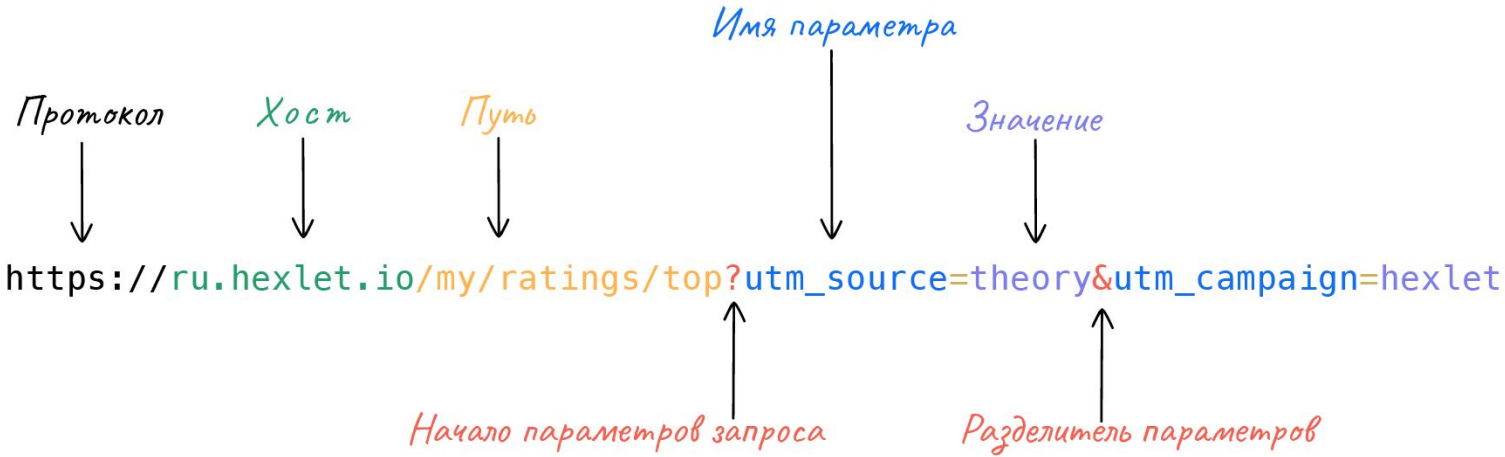
Для метода GET можно не указывать options

```
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 URL 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 URL.

```
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);
```

Ваша новая IT-профессия – Ваш новый уровень жизни

Программирование с нуля в
немецкой школе AIT TR GmbH

AIT
Hi-tech team