295 JS, API {REST}

Table des matières

[API-{REST} 2](#_Toc189571355)

[CRUD: 2](#_Toc189571356)

[HTTP réponse avec un code 2](#_Toc189571357)

[REGEX 2](#_Toc189571358)

[= == === 4](#_Toc189571359)

[Table JS 4](#_Toc189571360)

[Objects 4](#_Toc189571361)

[Bitwise Operators 4](#_Toc189571362)

[Npm 5](#_Toc189571363)

[Corrections 5](#_Toc189571364)

[Объекты (Objects) 5](#_Toc189571365)

[Массивы (Arrays) 6](#_Toc189571366)

[Таблицы (Массив объектов, JSON-данные) 6](#_Toc189571367)

[Функции (Functions) 6](#_Toc189571368)

[Циклы (Loops) 6](#_Toc189571369)

[Условия (If, Switch, Тернарный оператор) 7](#_Toc189571370)

[Промисы и Async/Await (Асинхронный код) 7](#_Toc189571371)

[Работа с DOM (Изменение HTML через JS) 8](#_Toc189571372)

[Классы (Объектно-Ориентированное Программирование, ООП) 8](#_Toc189571373)

[Fetch API (Работа с сервером) 8](#_Toc189571374)

[ Arrow function 9](#_Toc189571375)

[console.log 9](#_Toc189571376)

[Promise 9](#_Toc189571377)

[Sync/await 10](#_Toc189571378)

[Routes (Express.js) 10](#_Toc189571379)

[Requests 11](#_Toc189571380)

[ req.body 11](#_Toc189571381)

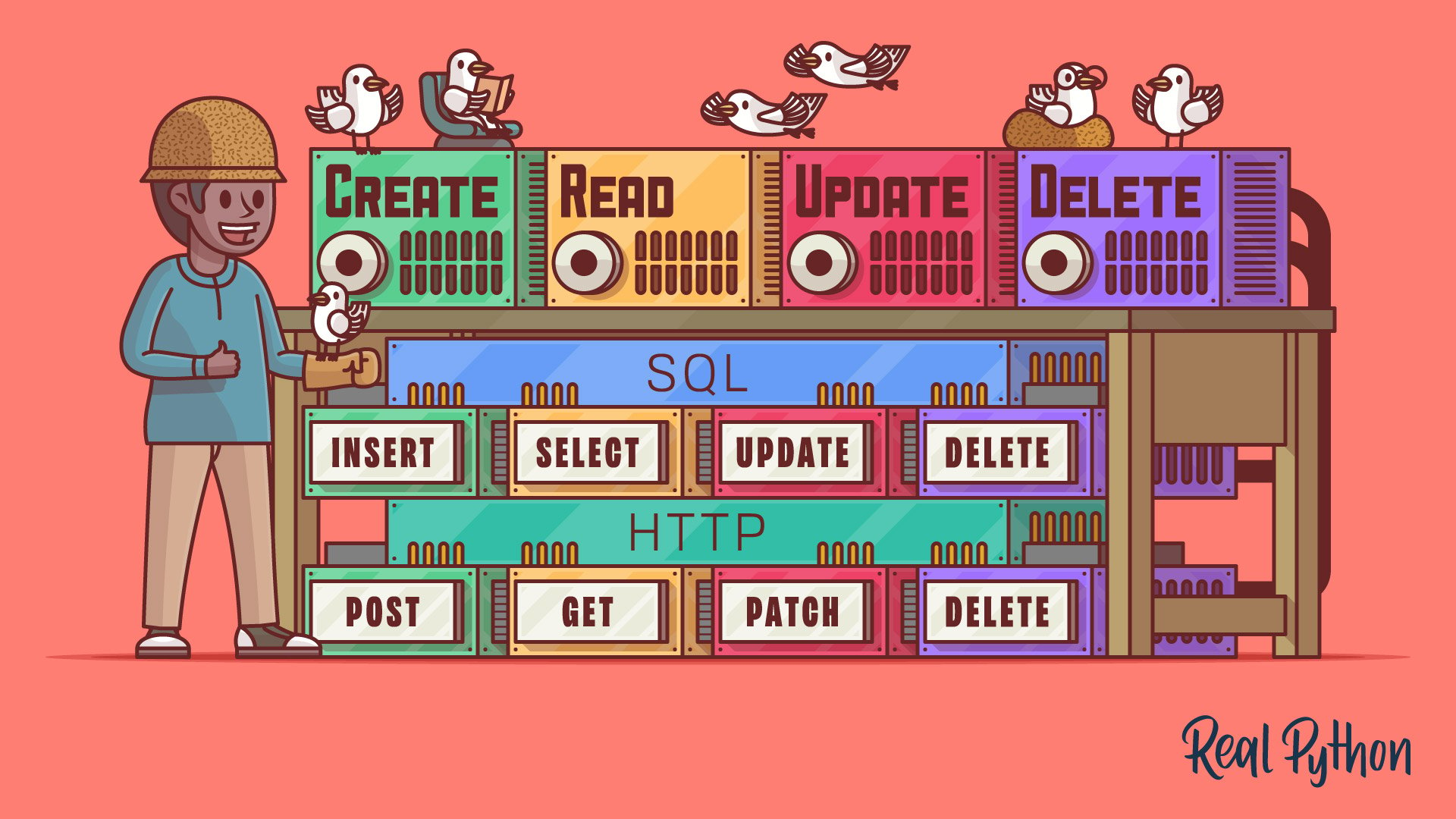
[ req.params 11](#_Toc189571382)

[ req.query 11](#_Toc189571383)

[Webographie 12](#_Toc189571384)

# API-{REST}

## CRUD:



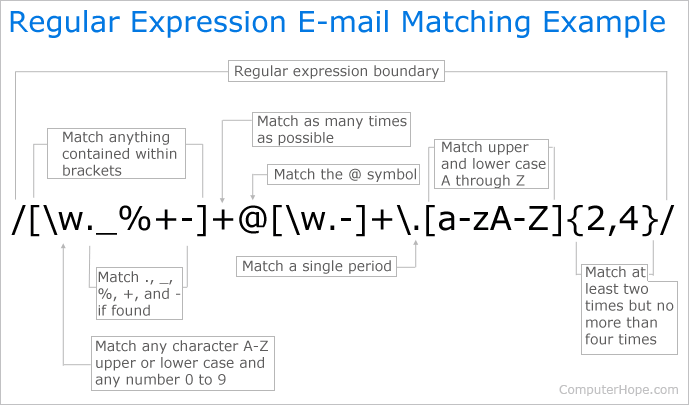
## HTTP réponse avec un code

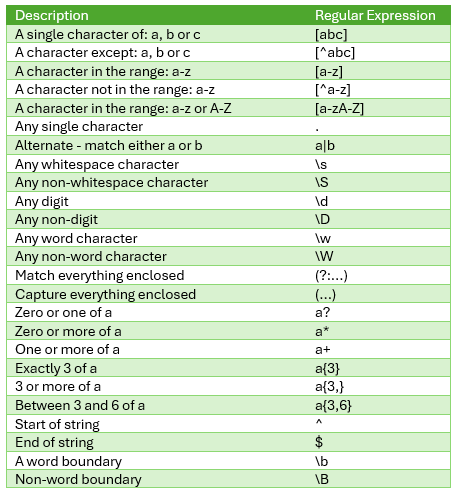
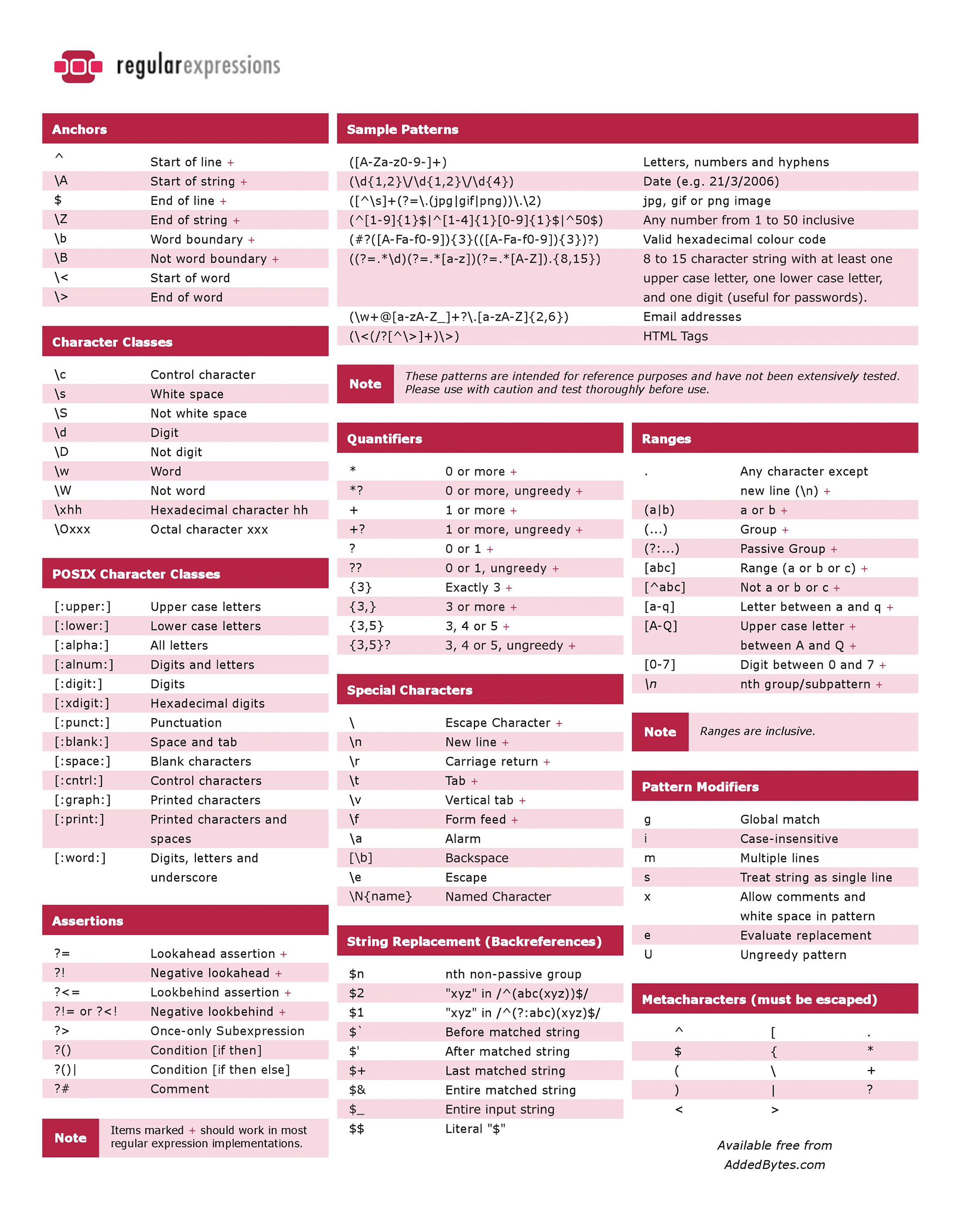
Lorsque vous envoyez une requête au serveur, le serveur vous renvoie une réponse avec un code d'état retour HTTP qui n'est pas visible dans votre navigateur.

**Chaque code est composé de 3 chiffres.**

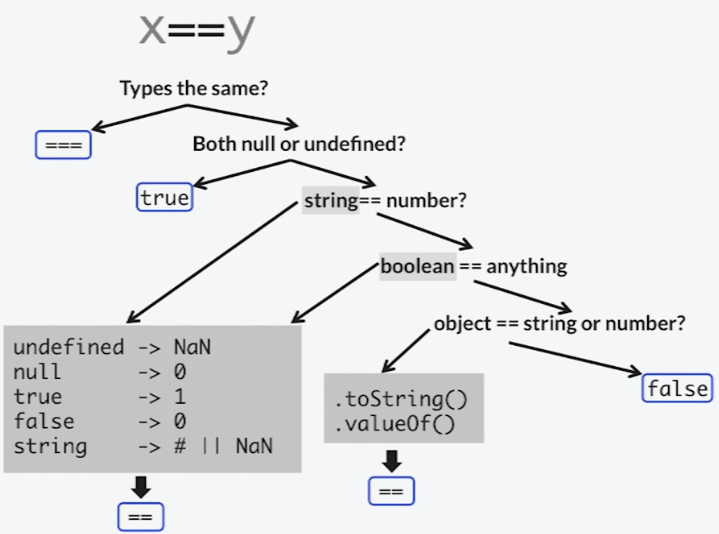
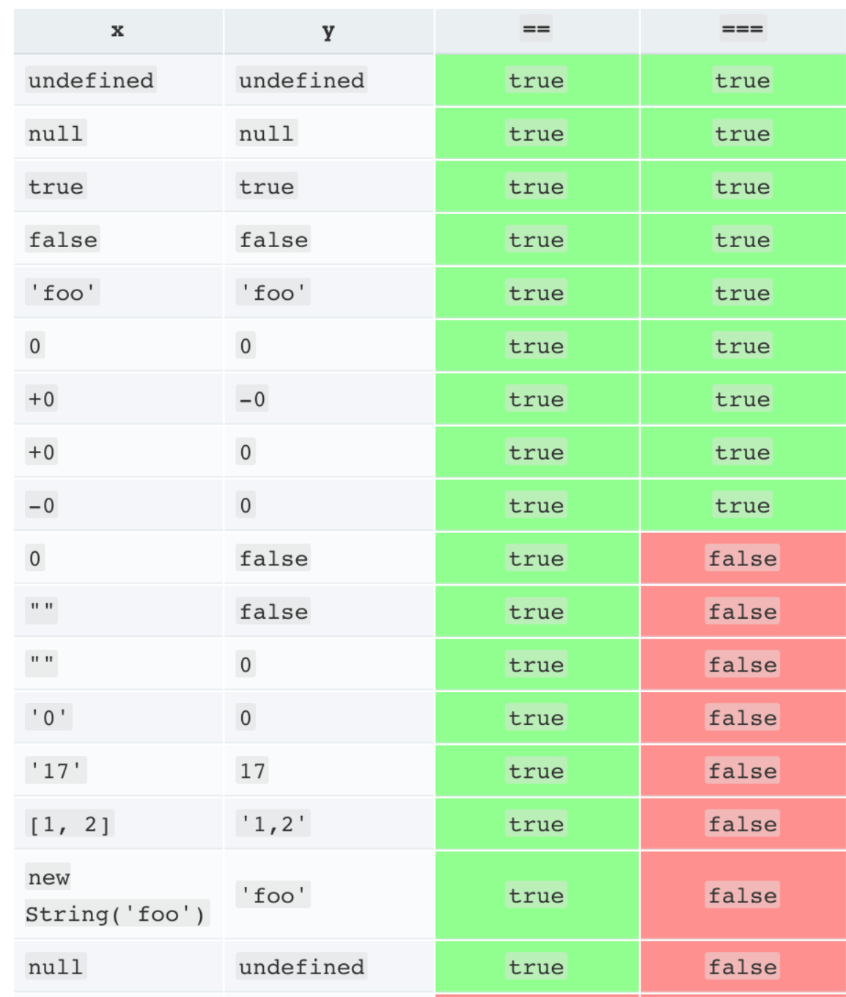
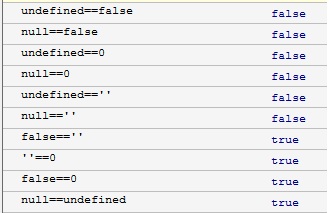
* **1xx : informatif :** communique une information au client ;
* **2xx : succès :** indique que la demande du client a été acceptée avec succès ;
* **3xx : redirection :** indique que le client doit prendre des mesures supplémentaires pour terminer sa demande ;
* **4xx : erreur du client :** indique une erreur de la part du client ;
* **5xx : erreur du serveur :** indique une erreur de la part du serveur.

## REGEX





# = == ===



**== (valeur)**

**===(valeur , type)**

## Table JS

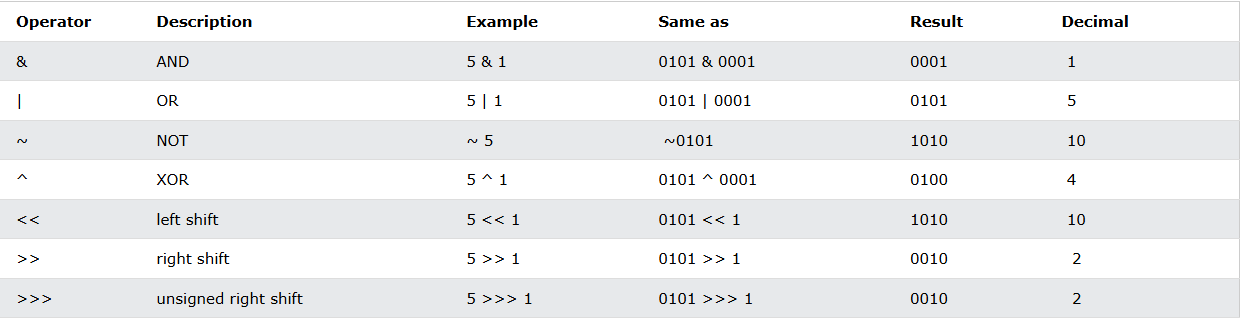
const products =[{id: 1, name: "Dan"},{id: 2, name: "Anne"},];

const table =[{a,b,c,},{a,b,c,}];

## Objects

const car = {type:"Fiat", model:"500", color:"white"};

## Bitwise Operators



## Npm

 *« package.json »*

* npm (Node Package Manager) — это менеджер пакетов для Node.js. Он позволяет устанавливать, обновлять и удалять зависимости, а также управлять пакетами и скриптами вашего проекта.
* **npm init** Создаёт файл package.json, который хранит информацию о проекте и его зависимостях.
* **npm install** или **npm i** Устанавливает зависимости, указанные в package.json.
* **npm update** Обновляет все зависимости проекта до последних версий, совместимых с указанными в package.json.
* **npm start** **(raccourci)** Запускает скрипт start, указанный в package.json. \*\**Fonctionne uniquement si* ***un script "start" est défini*** *dans* ***package.json****.\*\**
* **npm run start :** Fonctionne **toujours**, même si "start" n'est pas un script spécial. \*\**Peut être utilisé pour exécuter n'importe quel script défini dans* ***package.json****.*\*\*
* Pour la commande ***npm install*** quelle est la différence entre l'option ***--save*** et l'option ***--save-dev*** ?
* ***--save*** : il se mets dans package.json est ajouté dans"Dependencies".
* ***--save-dev*** : il se mets dans package.json est ajouté dan"devDependencies".

### Corrections

**nodemon** pour redemarer server autumatiquement (comme live server après chaque Ctrl +S).

# Объекты (Objects)

Объект – это структура, которая хранит **пары "ключ: значение"**.

let person = {

  name: "Yosef",

  age: 25,

  isStudent: true,

  sayHello: function() {

      console.log("Привет, меня зовут " + this.name);

  }};

console.log(person.name);  // "Yosef"

person.sayHello();         // "Привет, меня зовут Yosef"

✅ **Методы объектов** – это функции внутри объектов.  
✅ **this** используется для доступа к свойствам объекта.

## Массивы (Arrays)

Массив – это список значений.

let fruits = ["🍎", "🍌", "🍊"];

console.log(fruits[0]);  // "🍎"

console.log(fruits.length);  // 3

fruits.push("🍇");  // Добавить элемент

console.log(fruits);  // ["🍎", "🍌", "🍊", "🍇"]

✅**Методы массива:** .push(), .pop(), .shift(), .unshift(), .map(), .filter(), .reduce().

|  |  |
| --- | --- |
| .push() | Добавляет элемент в конец массива |

|  |  |
| --- | --- |
| .pop() | Удаляет последний элемент массива |

|  |  |
| --- | --- |
| .unshift() | Добавляет элемент в начало массива |

|  |  |
| --- | --- |
| .shift() | Удаляет первый элемент массива |

|  |  |
| --- | --- |
| .splice(start, deleteCount, ...items) | Удаляет, заменяет или добавляет элементы в массив |

|  |  |
| --- | --- |
| .slice(start, end) | Возвращает копию части массива |

## Таблицы (Массив объектов, JSON-данные)

Если нужно хранить **список объектов**, это называется **таблицей данных**:

let users = [

  { id: 1, name: "Yosef", age: 25 },

  { id: 2, name: "Anna", age: 30 }

];

console.log(users[1].name);  // "Anna"

✅ Часто используется для API и базы данных.

## Функции (Functions)

Функции могут быть **обычными** или **стрелочными**:

// Обычная функция

function sum(a, b) {

  return a + b;

}

// Стрелочная функция

let multiply = (a, b) => a \* b;

console.log(sum(5, 3));  // 8

console.log(multiply(5, 3));  // 15

## Циклы (Loops)

Циклы позволяют перебирать массивы и объекты.

let numbers = [10, 20, 30];

// for(let i=0; i<10; i++)

for (let i = 0; i < numbers.length; i++) {

    console.log(numbers[i]);  // 10, 20, 30

}

// forEach (только для массивов)

numbers.forEach(num => console.log(num));

// for...in (для объектов)

let person = { name: "Yosef", age: 25 };

for (let key in person) {

    console.log(key + ": " + person[key]);

}

// for...of (для массивов)

for (let num of numbers) {

    console.log(num);

}

✅ forEach() удобен для массивов.  
✅ for...in работает с объектами.  
✅ for...of лучше использовать для массивов.

## **Условия (If, Switch, Тернарный оператор)**

let age = 18;

if (age >= 18) {

    console.log("Совершеннолетний");

} else {

    console.log("Несовершеннолетний");

}

// Тернарный оператор (короткая запись)

let status = (age >= 18) ? "Взрослый" : "Ребёнок";

console.log(status);

// Switch-case

let fruit = "🍎";

switch (fruit) {

    case "🍎":

        console.log("Это яблоко");

        break;

    case "🍌":

        console.log("Это банан");

        break;

    default:

        console.log("Неизвестный фрукт");

}

✅ if...else для логики.  
✅ switch для множественных условий.  
✅ ? : (тернарник) – для коротких условий.

## Промисы и Async/Await (Асинхронный код)

В JavaScript код может быть **асинхронным** (например, запросы в базу или API).

// Промис

let fetchData = new Promise((resolve, reject) => {

  setTimeout(() => resolve("Данные загружены"), 2000);

});

fetchData.then(data => console.log(data));  // Выведет "Данные загружены" через 2 секунды

// Async/Await

async function getData() {

  let result = await fetchData;

  console.log(result);

}

getData();

✅ Promise и async/await нужны для работы с задержками и API.

## Работа с DOM (Изменение HTML через JS)

JavaScript позволяет изменять страницу:

document.getElementById("myText").innerHTML = "Привет, мир!";

Полный пример:

<!DOCTYPE html>

<html>

<head>

    <title>JS в HTML</title>

</head>

<body>

    <p id="myText">Текст</p>

    <button onclick="changeText()">Изменить</button>

    <script>

        function changeText() {

            document.getElementById("myText").innerHTML = "Новый текст!";

        }

    </script>

</body>

</html>

✅ document.getElementById("id") – изменение HTML-элементов.  
✅ onclick – обработка событий.

## Классы (Объектно-Ориентированное Программирование, ООП)

Как и в C#, в JavaScript есть **классы и объекты**:

class Car {

  constructor(brand, year) {

      this.brand = brand;

      this.year = year;

  }

  showInfo() {

      console.log(`Марка: ${this.brand}, Год: ${this.year}`);

  }

}

let myCar = new Car("Toyota", 2022);

myCar.showInfo();  // "Марка: Toyota, Год: 2022"

✅ class – создание класса.  
✅ constructor – конструктор класса.  
✅ this – обращение к свойствам объекта.

## Fetch API (Работа с сервером)

Если нужно отправить GET/POST-запрос на сервер:

fetch("https://jsonplaceholder.typicode.com/todos/1")

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

    .then(data => console.log(data));

✅ Используется для взаимодействия с API.

## Arrow function

(value) => { logic } //formule générale

const add = (a, b) => a + b;

**Если функция возвращает объект, нужно обернуть его в ():**

const getUser = (id) => ({ id: id, name: "Yosef" });

console.log(getUser(1)); // Выведет: { id: 1, name: 'Yosef' }

## console.log

console.log(` text ${variable}`); en C#: Console.Write($"{variable} text") ; **//permet d'insérer directement des variables dans une chaîne de caractères**

## Promise

**Promise** — это объект, который представляет результат асинхронной операции. Он может быть в одном из трёх состояний:

1. **Pending (ожидание)** — операция еще не завершена.
2. **Resolved (выполнено)** — операция завершена успешно.
3. **Rejected (отклонено)** — операция завершена с ошибкой.

**Promise** используется для работы с асинхронным кодом, чтобы избежать "адского коллбэка" (callback hell), делая код более понятным и удобным для чтения.

let promise = new Promise((resolve, reject) => {

  let success = true; // Попробуй изменить на false для ошибки

  if (success) {

    resolve("Операция успешна!");

  } else {

    reject("Произошла ошибка.");

  }

});

promise

  .then((result) => console.log(result))   // Вывод: Операция успешна!

  .catch((error) => console.log(error));   // Если error: Произошла ошибка.

 **.then()** — вызывается при успешном завершении операции.

 **.catch()** — вызывается, если произошла ошибка.

* **.finally()** — выполняется в любом случае, независимо от успешности.

## Sync/await

**async/await** — это синтаксический сахар, который упрощает работу с асинхронными операциями, используя Promise под капотом. Это позволяет писать асинхронный код так, как если бы он был синхронным, улучшая читаемость и облегчая обработку ошибок.

* **async** — ключевое слово перед функцией, чтобы она возвращала **Promise**.
* **await** — используется внутри **async** функции и приостанавливает выполнение до тех пор, пока **Promise** не вернёт результат.

// Функция, возвращающая Promise

function fetchData() {

  return new Promise((resolve) => {

    setTimeout(() => resolve("Данные загружены!"), 2000);

  });

}

// Функция, использующая async/await

async function getData() {

  try {

    let result = await fetchData(); // Ожидаем завершения fetchData

    console.log(result); // Вывод: Данные загружены!

  } catch (error) {

    console.error("Ошибка:", error);

  }

}

getData();

 **fetchData()** — возвращает Promise, который будет выполнен через 2 секунды.

 **await fetchData()** приостанавливает выполнение кода, пока Promise не завершится.

 Если всё прошло успешно, результат будет выведен в консоль.

## Routes (Express.js)

 GET – получение данных

 POST – отправка новых данных

 PUT / PATCH – обновление данных(PUT это - обновит все данные, a PATCH это только одну)

 DELETE – удаление данных

const express = require('express');

const app = express();

app.use(express.json()); // Позволяет работать с JSON-данными

// Пример маршрутов API

app.get('/users', (req, res) => {

    res.json([{ id: 1, name: 'Alice' }, { id: 2, name: 'Bob' }]);

});

app.post('/users', (req, res) => {

    const newUser = req.body;

    res.status(201).json({ message: 'Пользователь создан', user: newUser });

});

app.put('/users/:id', (req, res) => {

    const userId = req.params.id;

    res.json({ message: `Пользователь ${userId} обновлён` });

});

app.delete('/users/:id', (req, res) => {

    const userId = req.params.id;

    res.json({ message: `Пользователь ${userId} удалён` });

});

// Запуск сервера

app.listen(3000, () => console.log('Сервер запущен на http://localhost:3000'));

## Requests

req.body.name и req.params.id — это два разных способа доступа к данным в запросах **HTTP** в **Node.js**, особенно в **Express**

**req.body** — данные, отправленные в теле запроса (POST, PUT).

**req.params** — параметры из пути URL (например, /user/:id).

**req.query** — параметры, переданные в строке запроса (например, ?term=book).

**req.headers** — заголовки запроса, содержащие метаданные (например, User-Agent).

**req.cookies** — данные, сохраненные в куках, отправленных с запросом.

**req.method** — метод HTTP-запроса (GET, POST, PUT, DELETE и т.д.).

### req.body

req.body — это объект, который содержит данные, **переданные в теле запроса** {"name": "Yosef"} , обычно при отправке **POST** или **PUT** запросов. Это может быть форма, JSON или другие данные, отправленные с запросом.

* Обычно используется, когда клиент отправляет данные на сервер с помощью метода **POST** или **PUT**. Данные отправляются в теле запроса (например, при отправке формы, или через fetch или axios с **JSON**).

app.post('/submit', (req, res) => {

  const name = req.body.name; // Данные из тела запроса

  res.send(`Привет, ${name}!`);

});

### req.params

req.params — это объект**, содержащий параметры маршрута**. Когда вы определяете маршруты с динамическими частями (например, */:id*), Express автоматически помещает значения этих частей в req.params.

* + Используется в запросах типа **GET** или **DELETE**, когда URL включает переменные параметры. (GET /users/:id)

app.get('/user/:id', (req, res) => {

  const userId = req.params.id; // Получаем параметр из URL

  res.send(`Пользователь с ID: ${userId}`);

});

### req.query

req.query содержит строку запроса (*query string*) из URL, которая начинается после знака вопроса (**?**). Эти параметры обычно используются для фильтрации данных, поиска или передачи дополнительных настроек.

* + Используется когда нужно получить информацию о метаданных запроса, таких как тип контента (Content-Type), авторизационные данные (Authorization), язык и т.д.

app.get('/search', (req, res) => {

  const term = req.query.term; // Получаем значение из строки запроса

  const category = req.query.category;

  res.send(`Поиск по запросу: ${term}, категория: ${category}`);

});

### req.headers

req.headers содержит все заголовки, которые были отправлены вместе с запросом. Заголовки предоставляют информацию о запросе, такую как тип контента, тип устройства, авторизационные данные и т.д.

* + Используется когда нужно получить информацию о метаданных запроса, таких как тип контента (Content-Type), авторизационные данные (Authorization), язык и т.д.

 app.get('/info', (req, res) => {

   const userAgent = req.headers['user-agent']; // Получаем заголовок 'User-Agent'

   res.send(`Ваш браузер: ${userAgent}`);

 });

### req.cookies

req.cookies содержит все куки, которые были отправлены с запросом. Куки — это небольшие данные, которые сохраняются на стороне клиента и отправляются на сервер с каждым запросом.

* + Используется когда тебе нужно работать с данными, хранящимися в куках, например, для аутентификации или для хранения пользовательских предпочтений.

 app.get('/welcome', (req, res) => {

   const username = req.cookies.username; // Получаем имя пользователя из кук

   res.send(`Привет, ${username}!`);

 });

### req.method

req.method содержит HTTP метод, который был использован для отправки запроса (например, GET, POST, PUT, DELETE).

* Используется когда тебе нужно узнать, какой именно метод был использован для запроса, например, для обработки GET и POST запросов по-разному.

 app.all('/info', (req, res) => {

   res.send(`Метод запроса: ${req.method}`);

 });

# Webographie

<https://www.w3schools.com/jsref/jsref_map_get.asp>

<https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Array/reduce>

<https://chatgpt.com/c/672e2b2b-8fd0-800b-a0f5-ab8ae8515f66> JS-Cheatlist