

00. Особености на Judge системата на SoftUni

Тестваме в Judge дали ни иска като вход масив [] или функция с променливи

Пишем `console.log` в Judge и ако на изхода видим квадратни скоби, то трябва функцията да я напишем с масив
Това е начин да разбирам какви са входните данни за дадена задача

Judge работи като приема вход само на 1 функция. Т.е. ако има нужда от повече функции за разрешаване на дадена задача, трябва да ги сложим нестнати в една външна/обобщаваща функция.

```
function addAndSubtract(a, b, c) {
    function sum(numOne, numTwo) {
        return numOne + numTwo;
    }

    function subtract(num1, num2) {
        return num1 - num2;
    }

    let sumResult = sum(a, b);
    let finalResult = subtract(sumResult, c);
    console.log(finalResult);
}
```

Judge на SoftUni винаги подава на функцията в Node JS стрингове – затова е удобно да правим функциите си така:
ВАЖНО – реално можем да използваме само част от параметрите на функцията

```
function areaOfFigures(arg1, arg2, arg3) {
    let type = arg1;

    if (type === "square") {
        let side = Number(arg2);
        console.log(side * side);
    } else if (type === "rectangle") {
        let sideA = Number(arg2);
        let sideB = Number(arg3);
        console.log(sideA * sideB);
    } else if (type === "circle") {
        let r = Number(arg2);
        let result = Math.PI * r * r;
        console.log(result);
    } else if (type === "triangle") {
        let a = Number(arg2);
        let h = Number(arg3);
        console.log(a * h / 2);
    }
}
areaOfFigures("triangle", 5, 8);
```

За изпитите на Node JS трябва да използваме като вход масив/контейнер:

ВАЖНО: да не боравим директно/да не променяме директно `input` параметъра – **ако е референтен тип данни**

```

function workShop (input) {
    let numA = input[0];
    let numB = input[1];
    let name = input[2];
}
workShop(['20', '5678', 'Ivan Ivanov']);

```

По този начин с return да ги правим функциите когато е възможно

```

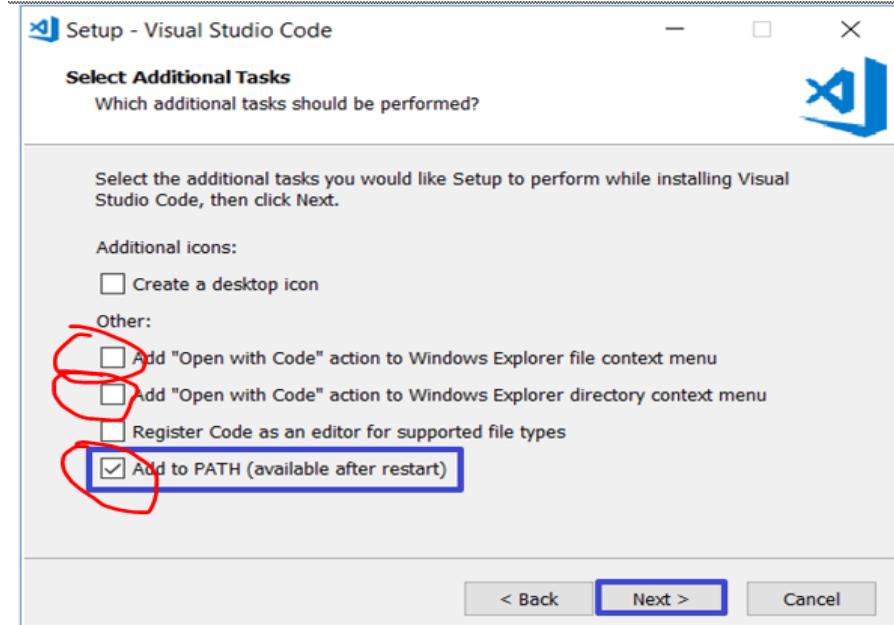
function solve(input = []) {
    let output = [];
    for (let i = 0; i < input.length; i++) {
        if (!output.includes(input[i])) {
            output.push(input[i]);
        }
    }
    return output;
}

console.log(
solve([7, 8, 9, 7, 2, 3, 4, 1, 2])
);

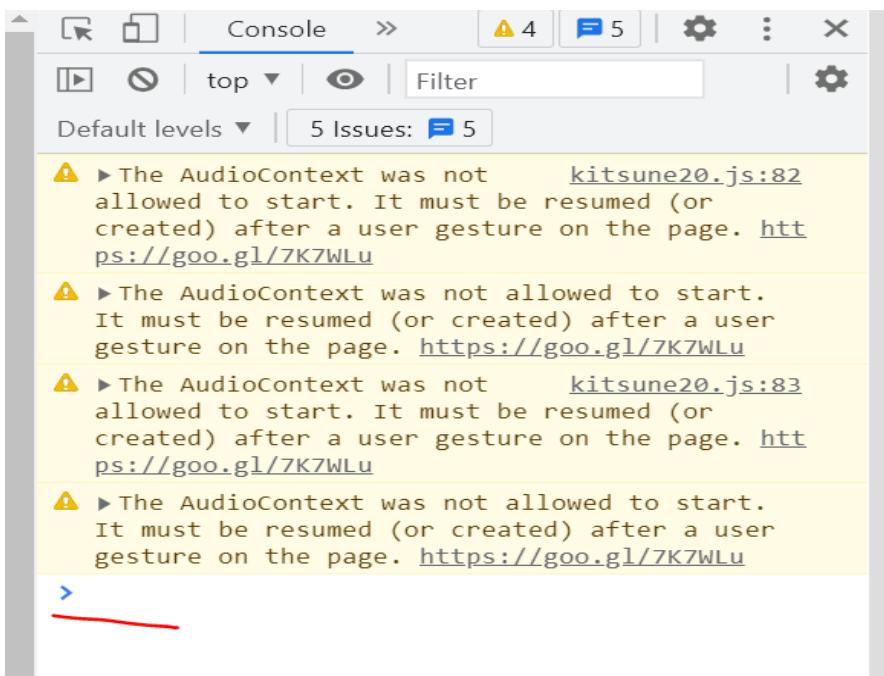
```

1. Introduction to JavaScript

1.1. IDE and configurations



Developer Console in Google Chrome - Във всеки един browser даваме **F12** и започваме да си пишем код
Developer Console Firefox Web Browser - **[Ctrl] + [Shift] + [i]**



В JS няма концепция да работи с конзола – той взема данните от средата на клиента, т.е. от браузъра!
JS е интерпретаторен език – ред по ред чете и го смята

Стартиране от терминала

The screenshot shows the VS Code interface with the 'TERMINAL' tab selected. A red circle highlights the 'TERMINAL' tab. The terminal window displays a PowerShell session:

```
Windows PowerShell
Copyright (C) Microsoft Corporation. All rights reserved.

Try the new cross-platform PowerShell https://aka.ms/pscore6

PS C:\Users\svilk\OneDrive\Soft Engineer\JavaScript-Node JS\Basic Module\05 - Simple operations & calculations> node yardGreening.js
The final price is: 3369.71 lv.
The discount is: 739.69 lv.
PS C:\Users\svilk\OneDrive\Soft Engineer\JavaScript-Node JS\Basic Module\05 - Simple operations & calculations>
```

The screenshot shows a Command Prompt window with the title 'Command Prompt - node'. The window displays a Node.js session:

```
Microsoft Windows [Version 10.0.19043.1165]
(c) Microsoft Corporation. All rights reserved.

C:\Users\svilk>node
Welcome to Node.js v12.16.2.
Type ".help" for more information.
> 5+5
10
>
```

JS test.js X

```
JS test.js
1 console.log(555);
2
```

TERMINAL DEBUG CONSOLE PROBLEMS OUTPUT

Windows PowerShell
Copyright (C) Microsoft Corporation. All rights reserved.

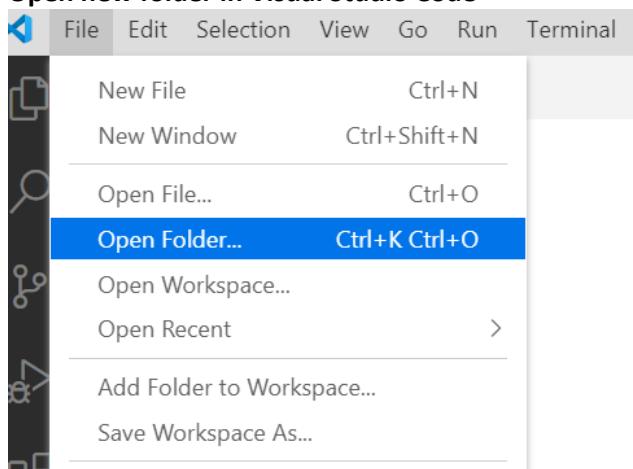
Try the new cross-platform PowerShell <https://aka.ms/pscore6>

PS C:\Users\svilk\OneDrive\Soft Engineer\JAVA & JS path\3_Node JS
- LAB> node test.js
555

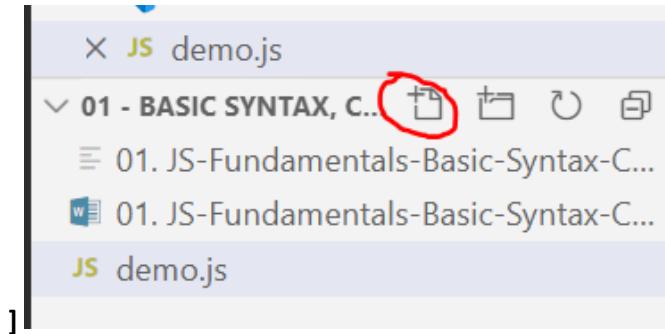
Using Visual Studio Code

Using WebStorm

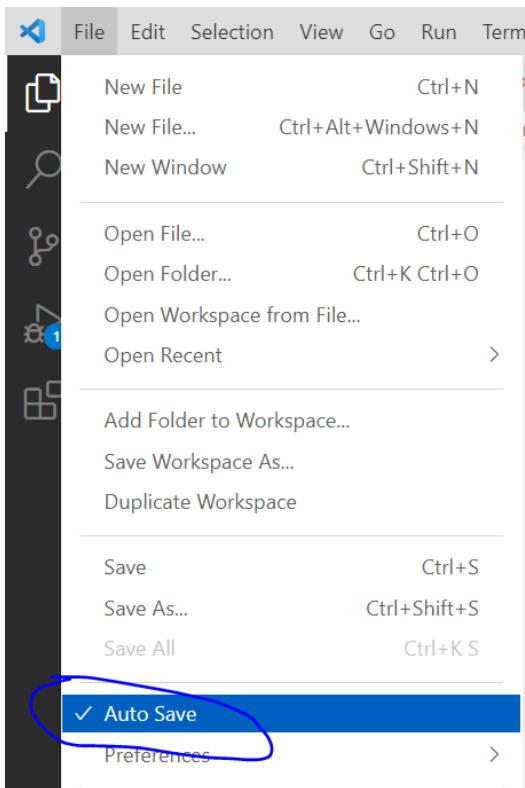
Open new folder in Visual Studio Code



New file



AutoSave mode ON



Настройка на JS – Preferences -> Settings

Editor: Tab Completion
Enables tab completions.
on

A screenshot of the 'Settings' section for 'Editor: Tab Completion'. It shows the setting name 'Editor: Tab Completion', a description 'Enables tab completions.', and a dropdown menu with the value 'on' selected.

Работа като на MS DOS от Terminal-а на Visual Studio Code

TERMINAL DEBUG CONSOLE PROBLEMS OUTPUT powershell + ls

```
Windows PowerShell
Copyright (C) Microsoft Corporation. All rights reserved.

Try the new cross-platform PowerShell https://aka.ms/pscore6

PS C:\Users\svilk\OneDrive\Soft Engineer\JAVA & JS path\3_Node JS core\2 - Fundamentals\prepare - January 2020 course\05 - Arrays - LAB> ls

Directory: C:\Users\svilk\OneDrive\Soft Engineer\JAVA & JS path\3_Node JS core\2 - Fundamentals\prepare - January 2020 course\05 - Arrays - LAB

Mode LastWriteTime Length Name
---- -- -- -- -- -
-a---l 7.9.2021 г. 22:09 215 01_SumFirstLastArrayElements.js
-a---l 7.9.2021 г. 22:24 336 02_DayOfWeek.js
```

A screenshot of the Visual Studio Code terminal window. The title bar says 'TERMINAL'. The bottom status bar shows 'powershell' and 'ls'. The terminal output shows a Windows PowerShell prompt, copyright information, a link to the new cross-platform PowerShell, and a directory listing for 'LAB' files. A red arrow points to the 'ls' command in the terminal.

settings.json

```
1. {
2.     "debug.javascript.autoAttachFilter": "always",
3.     "editor.fontSize": 18,
4.     "launch": {
```

```

5.     "version": "0.2.0",
6.     "configurations": [
7.       {
8.         "type": "node",
9.         "request": "launch",
10.        "name": "Launch Program",
11.        "program": "${file}",
12.        "cwd": "${workspaceRoot}",
13.        "outputCapture": "std"
14.      }
15.    ],
16.    "compounds": []
17.  },
18.  "workbench.iconTheme": "material-icon-theme",
19.  "[javascript)": {
20.    "editor.defaultFormatter": "esbenp.prettier-vscode"
21.  },
22.  "prettier.singleQuote": true,
23.  "[html)": {
24.    "editor.defaultFormatter": "esbenp.prettier-vscode"
25.  },
26.  "prettier.tabWidth": 4
27. }

```

Workbench (6)

- Appearance (3)**
- Breadcrumbs (1)
- Editor Management (1)

Features (1)

- Terminal (1)

Workbench: Icon Theme
Specifies the file icon theme used in the workbench or 'null' to not show any icons.

Seti (Visual Studio Code)

Workbench: Color Theme
Specifies the color theme used in the workbench.

1.2. JavaScript Syntax

! Нямаме типове на променливите

Инициализиране на променлива

```
let greeting = "Hello, " + name;
var
let variableName camelCase
```

Функция

Функциите в JS са **camelCase** – като в Java

Правило за скобите

```
function addNumbers(firstNumber, secondNumber) { - параметри firstNumber и secondNumber
  console.log(firstNumber + secondNumber);
}
addNumbers(1, 5); - аргументи 1 и 5
```

arguments is an Array-like object accessible inside [functions](#) that contains the values of the arguments passed to that function.

```
function addNumbers(firstNumber, secondNumber) {  
    return firstNumber + secondNumber;  
}  
console.log(addNumbers(1, 5));
```

Точка и запетая на края на реда не са задължителни в JS

JS не се компилира, а само се интерпретира от Node JS

Принтиране

```
function Hello() {  
    console.log("Hello, SoftUni");  
}  
Hello();
```

Печата на същия ред няколко неща

```
console.log("Hello, SoftUni", "We are super"); // Hello, SoftUni We are super
```

Бързи команди в Visual Studio Code:

MDN JavaScript – всичко за Java Script в интернет - <https://developer.mozilla.org/en-US/docs/Web/JavaScript>

Като посочим с курсора дадена функция, и ни излиза какво точно прави

F2 върху маркираното – сменя името навсякъде на тази променлива/метод/функция

Ctrl + click върху функция – дава препрадка към държавната библиотека

Alt + Shift + F – форматира автоматично

Ctrl + F5 = Run without debugging in Visual Studio Code

Ctrl + / - слага и маха го в коментар

Ctrl + K + C и Ctrl + K + U - слага и маха го в коментар

Пишем log + Enter и се изписва console.log автоматично

Ctrl + D – първо маркираме даден текст/запетая който искаме да махнем, и с Ctrl + D селектираме всеки следващ такъв елемент/текст/запетая в една обща селекция

Shift + Alt + I – селектира много редове, като с Ctrl + стрелка наляво отива в началото (съответно Ctrl + стрелка надясно)

Когато не е маркиран реда, то с Ctrl + C го копираме

Alt + Shift + местена на мишката – селектира много редове да се пише едновременно

Alt + стрелка нагоре/надолу – местим даден ред нагоре/надолу

Ctrl+Shift+Space докато курсора е въртре в скобите на някой метод – показва какво прави функцията

Split Right – когато искаме да гледаме едновременно както .js файла, така и .html файла.

Tab – мести селектирани редове надясно

Shift + Tab - мести селектирани редове наляво

Ctrl + Space при импортиране на функции

IntelliSense

За да засича Visual Studio Code дадена функция, то променливата трябва да е ясно декларирана от какъв тип е! При WebStorm IDE ще се усети лесно – по-интегрирана, но и по-тежка програма!

Т.е. за да работи IntelliSense, трябва изрично да задаваме типа на дадена променлива в параметрите на функцията – идва от ECMA script 6

The screenshot shows a code editor window for '01_AddSubtract.js'. A tooltip is displayed over the parameter 'data' in the line 'function solve(data = []) {'. The tooltip lists several array methods: concat, copyWithin, entries, every, fill, filter, find, findIndex, flat, flatMap, forEach, and includes. The 'concat' method is highlighted.

```
1 function solve(data = []) {
2     data.
3 }
```

- concat
- copyWithin
- entries
- every
- fill
- filter
- find
- findIndex
- flat
- flatMap
- forEach
- includes

По този начин му задаваме от какъв тип е параметъра input и така работи IntelliSense

Пишем /** и му даваме enter на това което ни излиза, и се получава следното:

```
/**  
 *  
 * @param {string} input  
 */  
  
/**  
 *  
 * @param {array} input  
 */  
  
function echo(input) {  
    console.log(input.length);  
    console.log(input);  
}  
  
echo('Hello Java Script');
```

```
## table geneartion  
/**  
 *  
 * @param {Event} e  
 */  
function generate(e) {  
}
```

```
/**  
 *
```

```

* @param {function} area
* @param {function} vol
* @param {string} input
*/
function solve(area, vol, input) {
    // console.log(JSON.parse(input));
    const cubes = JSON.parse(input);
    const cube = cubes[0];

    const cubeArea = area();
}

```

Понякога, след като напиша точка, трябва ръчно да го избера и натисна, защото с copy-paste интерпретаторът на Visual Studio Code не го разбира, въпреки че е правилно написано.

Тагове за нов ред

Тези 3 да не ги ползваме

\n
\r\n
\r

`console.log(output.join("\n"));` - отпечатва обединено на нов ред

Оператори за сравнение в JS

Оператор	Означение
Равенство по стойност (и тип данни)	<code>==, ===</code>
Различно по стойност (и тип данни)	<code>!=, !==</code>
По-голямо	<code>></code>
По-голямо или равно	<code>>=</code>
По-малко	<code><</code>
По-малко или равно	<code><=</code>

Operator	Description	Example
<code>==</code>	equal to (no type)	<code>if (day == 'Monday')</code>
<code>></code>	greater than	<code>if (salary > 9000)</code>
<code><</code>	less than	<code>if (age < 18)</code>
<code>====</code>	equal to (with type)	<code>if (5 === 5)</code>
<code>>=</code>	greater than or equal (no type)	<code>if (6 >= 6)</code>
<code>!==</code>	not equal (with type)	<code>if (5 !== '5')</code>
<code>!=</code>	not equal (no type)	<code>if (5 != 5)</code>

```
console.log(2 == "2"); // връща true
console.log(2 === "2"); // връща false, повечето сравнения в JS са с 3 пъти равно
```

Type coercion in JS

Вид конвертиране, което позволява на JS автоматично да си реши кое от двата типа променливи да го приравни към еднакъв тип, за да може да направи сравнението.

`console.log(2 == "2");` // връща true – прави сравнението с coercion – привежда ги към един и същи тип, за да може да стане операцията

`console.log(2 === "2");` // връща false – блокира coercion тройното равно, повечето сравнения в JS са с 3 пъти равно

Сравняване с 3 променливи

```
if (field[0][y] == field[1][y] == field[2][y]) {
    return true;
}
```

Оператори за условност:

"`&&`" - И

"`||`" - ИЛИ

`^` - изключващо Или (XOR)

`!` – отрицание `if (!(command === "Done")) {}`

Nullish coalescing operator (`??`)

The nullish coalescing operator (`??`) is a logical operator that returns its right-hand side operand when its left-hand side operand is null or undefined, and otherwise returns its left-hand side operand.

The nullish coalescing operator has the fifth-lowest [operator precedence](#), directly lower than `||` and directly higher than the [conditional \(ternary\) operator](#).

```
const foo = null ?? 'default string';
console.log(foo);
// expected output: "default string"
```

```
const baz = 0 ?? 42; // 0 is falsy value
console.log(baz);
// expected output: 0
```

Тернарен оператор

```
console.log((year % 4 === 0 && year % 100 !== 0) || (year % 400 === 0) ? 'yes' : 'no');
```

Conditional Statements

В JS можем освен true И false, да имаме в условни конструкции if ("some text")

```
let a = 5;
if (a >= 5) {
    console.log(a);
} else {
    console.log('no');
}
```

Условни конструкции – като на Java:

```
if (condition) {
} else {
}
```

Ако конструкция if няма скоби, се изпълнява само следващият ред – като в Java

Switch

```
switch (a) {
    case 1: console.log("1");
    break;
    case 2: console.log("2");
    break;
    default: console.log("Error");
    break;
}
```

И за щатите и за Англия, все е English

```
switch (country){
case 'USA':
case 'England':
    console.log('English');
    break;
}
```

Loops - Цикли:

```
for (let i = 1; i <= 10; i++) {
    console.log(i);
}
```

```
let a = 5;
while (a <= 10) {
    console.log("a = " + a);
    a++;
}
```

```
}
```

- Оператор **break** – прекъсва цикъла

```
while (true) {
    if (...) {
        break;
    }
}
```

- Оператор **continue** – преминава към следващата итерация на цикъла

```
for (let i = 0; i < 10; i++) {
    if (i % 2 === 0) {
        continue;
    }
    console.log(i);
}
```

Date:

Syntax

```
new Date()
new Date(value)
new Date(dateString)
```

```
new Date(year, monthIndex)
new Date(year, monthIndex, day)
new Date(year, monthIndex, day, hours)
new Date(year, monthIndex, day, hours, minutes)
new Date(year, monthIndex, day, hours, minutes, seconds)
new Date(year, monthIndex, day, hours, minutes, seconds, milliseconds)
```

```
const date = new Date(year, month-1, day);
let dateNumberValueInMilliseconds = date.valueOf();
```

```
console.log(date); // Fri Sep 30 2016 00:00:00 GMT+0300 (GMT+03:00)
console.log(dateNumberValueInMilliseconds); // 1475182800000
```

```
let dateString = year + "-" + month + " " + day;
let event = new Date(dateString);
event.setDate(day-1);
```

```
console.log(event.getFullYear() + ' ' + Number(event.getMonth() + 1) + ' ' + event.getDate())
;
```

1.3. Debugging the Code

Using the Debugger in Visual Studio Code

- Visual Studio Code has a built-in **debugger**
- It provides:

- **Breakpoints**
- Ability to **trace** the code execution
- Ability to **inspect** variables at runtime

Посочваме с курсора и то ни показва текущата стойност.

Дебъгване:

F5 - Start Debugging

F5 - Continue

F10 – Step over – trace step by step – ред по ред

F11 – step into – влиза в дадена функция / стек

Shift + F11 – step out – излиза от дадена функция / стек

F9 – stoppers - toggle a breakpoint – слага breakpoint на реда, на който е курсорът

Shift + F11 – директно отивам до stopper



Run & Debug



The screenshot shows the WebStorm IDE interface during a debugging session. The left sidebar contains several sections:

- VARIABLES**: Shows a tree structure with **Block: login** containing **i: 3** and **this: global**; and **Local: login** containing **failsCounter: 3**, **input: (4) ['login', 'go', 'l...', 'password: 'recA'**, **username: 'Acer'**.
- WATCH**: Empty.
- CALL STACK**: Shows a single entry: **Launch Program: 09_Lo... PAUSED**. Below it, there are two frames:
 - login** at line 6:40 in **09_Login.js**
 - <anonymous>** at line 24:1 in **09_Login.js**A link to "Show 6 More: Skipped by skipFiles" is also present.
- LOADED SCRIPTS**: Empty.
- BREAKPOINTS**: Empty.

The main editor area displays the **09_Login.js** file with the following content:

```
JS 09_Login.js X
JS 09_Login.js > login
1  function login(input) {
2      let username = input.shift();
3      let password = username.split('').reverse().join('');
4      let failsCounter = 0;
5
6      for (let i = 0; i < input.length; i++) {
7          let current = input[i];
8
9          if (current !== password) {
10              failsCounter++;
11              if (failsCounter > 3) {
12                  console.log(`User ${username} blocked!`);
13                  break;
14              }
15              console.log('Incorrect password. Try again.');
16          } else {
17              console.log(`User ${username} logged in.`);
18          }
19      }
20  }
21
22 }
```

The line **for (let i = 0; i < input.length; i++) {** is highlighted in yellow, indicating it is the current line being executed. A red dot marks the breakpoint at line 7. The terminal below shows the output of the program:

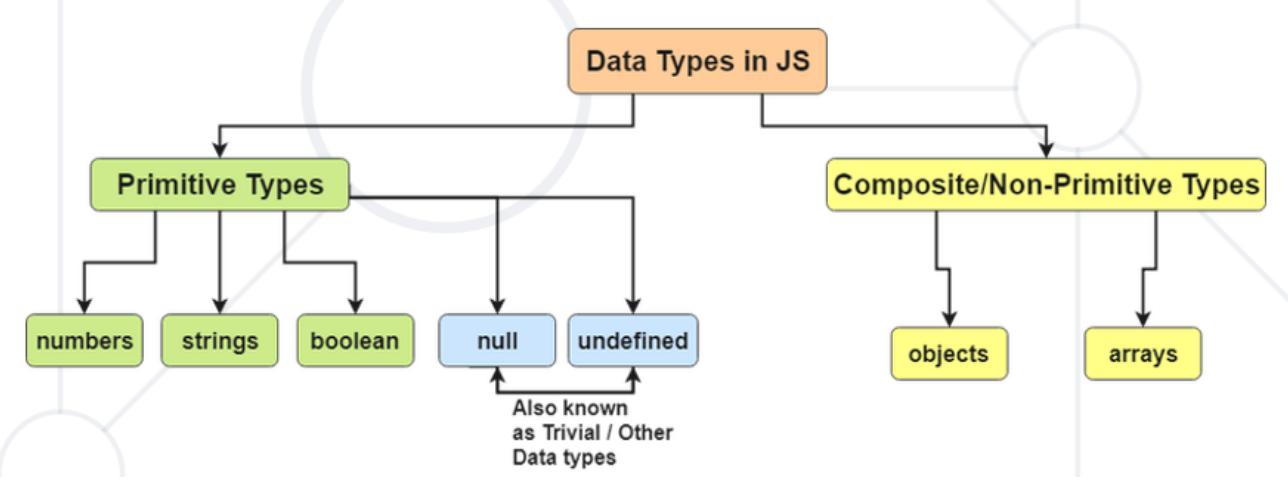
```
C:\Program Files\nodejs\node.exe .\09_Login.js
3 Incorrect password. Try again.
User Acer logged in.
```

Using the Debugger in WebStorm

2. Data Types and Variables

2.1. Definition

The **data type** of a value is an attribute that tells what kind of data that value can have



```

let number = 10;           // Number
let name = 'George';       // String
let array = [1, 2, 3];      // Array
let isTrue = true;          // Boolean
let person = {name: 'George', age: 25}; // Object
let empty = null;           // Null
let unknown = undefined;    // Undefined
  
```

ECMAScript 2015 – на различен браузър по различен начин върви кода – фирмата Ecma International решава какви изисквания JS трябва да имплементира, за да може скрипта да работи на всеки браузър. - стандарти

JavaScript is a **dynamic** language

Variables are **not** directly associated with any particular value type

Any variable can be **assigned** (and **re-assigned**) values of all types:

Литерал – начина, по който инициализираме дадена променлива казва какъв ще е типът на тази променлива

```

let variable = 42; // variable is now a number
variable = 'bar'; // variable is now a string
variable = true; // variable is now a Boolean
variable = 3.14; // variable is now a number
  
```

2.2. Let vs. Var - Local vs. Global

Деклариране

Инициализиране

- **var** - variables declared inside a block {} can be accessed from outside the block – използвало се масово до 2015 година

```

{ var x = 2; }
console.log(x); // 2
  
```

- **let** - variables declared inside a block {} can NOT be accessed from outside the block

```

{ let x = 2; }
console.log(x) // undefined
  
```

Variables Scope - The scope of a variable is the region of the program in which it is defined

- **Global Scope** – Global variables can be accessed from anywhere in a JavaScript function

```
var carName = "Volvo"; // Code here can use carName
function myFunction() { // Code here can also use carName
}
```

Където и да се намираме, ако не декларираме с var или let, то променливата става глобална! – да не го използваме

```
function myFunction() {
    carName = "Volvo";
    //Here code CAN use carName
}
//Here code also CAN use carName
```

- **Function Scope** – Local variables can only be accessed from inside the function where they are declared

```
function myFunction() {
    var carName = "Volvo";
    // Only here code CAN use carName
}
```

- **Block Scope** - Variables declared inside a block {} can not be accessed from outside the block

let – scope се ограничава до блок scope

```
{ let x = 2;
// x can NOT be used here
```

const – scope се ограничава до блок scope

```
{
    const x = 2;
    x = 5; //дава грешка, не може константа да ѝ променяме стойността
    console.log(x);
}
// x can NOT be used here
```

Naming Variables – we use camelCase in 99% of the cases!!!

- Variable names are **case sensitive**
- Variable names must begin with a **letter** or **underscore** (_) character
- Variable names **can't** be one of JavaScript's reserved words like: **break, const, interface, typeof, true** etc.

2.3. Strings - Sequence of Characters

В JS стринга е immutable

В JS нямаме char / Character !!!

Тилда кавички `` - за интерполяция

Единични кавички “ ”

Двойни кавички “”

- Used to represent **textual data**

- Each **symbol** occupies a **position** in the String
- The **first** element is at **index 0**, the next at index 1, and so on
- The **length** of a String is the number of elements in it

```
let name = 'George';
console.log(name[0]); // 'G'
console.log(name.length); // 6
```

Strings Are Immutable – like in JAVA

- Unlike in languages like C, JavaScript strings are **immutable**
- This means that once a string is created, it is **not** possible to **modify** it

```
let name = 'George';
name[0] = 'P';
console.log(name) // 'George'
```

`arr.length` – дължината на масива/дължината на string-а/низ-а

Сравняваме Стинг също със знак равно

```
if (role !== "Administrator") {
    console.log("No permission");
}
```

Всичките букви малки / големи

```
let a = "caseSensitive".toLocaleLowerCase; toLocaleUpperCase
toLowerCase; toUpperCase
```

Работа с текст / String

Няма значение дали единични или двойни кавички използваме. Една от причините да се използват в практиката повече единични кавички, е че да се отличават от двойните кавички на HTML атриутите

Когато има нужда да използваме единия вид кавички за част от думата, то използваме другия вид кавички

```
console.log("Kv'o stava"); // Kv'o stava
console.log('Kv\'o stava'); // Kv'o stava – работи с escape
```

```
console.log('Ami " Sega'); // Ami " Sega
```

`console.log(5)` вика `toString()` метода автоматично

```
let text = 'SoftUni';
let length = text.length; // 7
```

```
let text = 'SoftUni';
let letter = text[4]; // връща буква U

for (let i = 0; i < text.length; i++) {
    console.log(text[i]);
```

```
}
```

.trim() - Премахване на празни интервали/white spaces отпред и отзад на даден стринг -

Конкатенация

```
function greeting(name){  
    console.log("Hello, " + name + "!");  
}  
  
toString() и includes()  
let sum = 1563;  
let sumString = sum.toString(); //прави го от число на String  
let numString = 1563 + + ''; //също го прави от число на String  
  
sumString.includes(9); //дали String-а включва цифрата 9
```

Статично от ASCII код номер към символ

```
String.fromCharCode(97, 98, 99); //връща abc
```

От даден символ към ASCII код номер

```
let firstNum = 'a'.charCodeAt(0);
```

Replace функция

```
let matchWord = firstWord.replace('_', character); //замени долна черта с character само за първия намерен такъв символ
```

.slice() функция при стрингове

```
let color = 'red';  
color[0].toUpperCase() + color.slice(1, color.length); // Red
```

копира целият стринг, без последния символ

```
let sss = "Hello";  
console.log(sss.slice(0, sss.length-1)); //маха последния символ  
console.log(sss.slice(0, -1)); //маха последния символ с използване на знак минус
```

Изпечатва колко променливи има зададени на функцията (4ри в случая), но използваме само 3 параметъра

```
function rightPlaceWithReplace(firstWord, character, secondWord) {  
    console.log(arguments);  
  
    let matchWord = firstWord.replace('_', character);  
  
    if (matchWord === secondWord) {  
        console.log("Matched");  
    } else {
```

```

        console.log("Not Matched");
    }
}
rightPlaceWithReplace('Str_ng', 'I', 'Strong', 'zzzz');

```

При JS системата ни позволява да извикваме индекс извън колекцията. При parse на този елемент, връща NAN

2.4. String Interpolation

In JS we can use **template literals**. These are string literals that allow **embedded** expressions

Шаблони - **Интерполация** със стринг се извършва с **тилда кавички** и се реферира чрез `${firstName}`

```

function concatenate(firstName, lastName, age, town) {
    console.log(`You are ${firstName} ${lastName}, a ${age}-years old person from ${town}.`);
}
concatenate("Todor", "Stoyanov", "34", "Shumen");

```

Използване на функцията repeat

```

function triangle(number) {
    for (let i = 1; i <= number; i++) {
        console.log(`${i}`.repeat(i));
    }
}

```

2.5. Numbers

JS Node се старае когато има аритметични операции, да преобразува в число вместо нас!

Integer, Float, Double – All in One

- There is **no specific** type for integers and floating-point numbers
- To represent floating-point numbers, the number type has three symbolic values: **+Infinity**, **-Infinity**, and **NaN** (not-a-number)

Литерал за безкрайност:

```

let a = Infinity;
console.log(10 < a); //true

```

```

let num1 = 1;
let num2 = 1.5;
let num3 = 'p';
console.log(num1 + num2)           // 2.5
console.log(num1 + num3)           // '1p'
console.log(Number(num3))         // NaN not-a-number

```

Прави го на число

```

let text = "25.7";
let result = Number(text); - прави го на цяло или на дробно число
let result = +(text); - прави го на цяло или на дробно число

```

```
let text = "10Pesho";
let result = Number(text); //връща NaN
```

Деление на числа (оператор /):

```
let a = 25;
let i = a / 4;           // 6.25 (дробно делене връща)
let result = parseInt(6.25); // 6 (дробната част се отрязва)
let result = parseInt("10Pesho"); // връща 10 !!!
let f = parseFloat(a / 4.0); // 6.25 – връща реално число
let infinity = a / 0;      // Infinity (безкрайност)
let sqrt = Math.sqrt(-1); // получава се NaN = Not a Number
```

Модул/остатък от целочислено деление на числа (оператор %):

```
let product = 7 % 2; //връща остатъка от делението
```

Закръгления

```
let up = Math.ceil(23.45); // up = 24 – връща цяло число
let down = Math.floor(45.67); // down = 45 – връща цяло число
console.log(Math.round(5.55)); //6
console.log(Math.round(5.45)); //5
```

```
let trunc = Math.trunc(45.67); // trunc = 45 – изрязва дробната част, като parseInt
```

В JS има деление с остатък, но няма целочислено деление като при JAVA (11 / 2 дава винаги 5,5). Затова използваме функцията **Math.trunc** – да изрежем дробната част и така да получим цялата част. Или използваме **Math.floor**. Или използваме **parseInt**.

```
let number = Math.trunc(number/2);
```

Делене и връщане на цяло число:

```
let number = Math.trunc(number/2); - все едно деленото в Java, което връща цяло число
let number = parseInt(number/2); - все едно деленото в Java, което връща цяло число
```

Форматирания – до 2рия знак след десетичната запетая – връща String

(123.456).toFixed(2); // 123.46 - връща String като закръгля нормално до 2рия знак – в случай че има други цифри след 2-рата след десетичната запетая

или

```
console.log(`$(sales * commission).toFixed(2)');
```

toFixed(0) – връща стринг до закръглено до цяло число

toFixed(2) – връща стринг дробно число, закръглено до втората цифра след десетичната запетая

```
console.log(Number(Number('36.502').toFixed(2))); // 36.5
```

```
console.log(Number('36.502').toFixed(2)); //36.50
```

isNaN(3586.47) – функция, която проверява дали е not-a-number

В console.log не може да слагаме операции `+*/` - трябва да си направим предварително променлива, в която да сметнем резултата, и резултата да разпечатаме

```
function solve() {  
    let a = 25;  
    let result = a / 4;  
    console.log(result);  
}
```

Когато само декларираме променлива, но не и зададем стойност.

```
let firstDay;  
if (firstDay === undefined) {  
    firstDay = i+1;  
}  
  
console.log(2 + 5 + "5" + 10); - връща 7510
```

`console.log(0.1 + 0.2);` - връща 0.3000000000000004 – във всички езици го има това, на ниво процесор е проблема, **IEEE 754** - IEEE Standard for Floating-Point Arithmetic – има разлика след десетичната запетая/губим данни

```
Math.abs();  
Math.PI();  
Math.floor(); // закръгля към по-малкото цяло число – като truncate и като parseInt  
Math.ceil();  
  
let minDiff = Number.MAX_SAFE_INTEGER;
```

От двоична в десетична система – по краткия начин

```
let number = '1101';  
parseInt(number, 2); //13
```

От десетична в двоична система – по краткия начин

```
let num = 2;  
num.toString(2); // "10"
```

От 16чно число към десетично

```
yourNumber = parseInt(hexString, 16);
```

От 10чно число към 16чно число

```
hexString = yourNumber.toString(16);
```

С нули отпред:

```
var num = "000" + 123  
"123".padStart(5, 0)
```

Дали е цяло число проверка

```
Number.isInteger(distance(x1, y1, 0, 0))
```

```
let biggest = Number.MIN_SAFE_INTEGER;
```

2.6. Booleans - conditions

- **Boolean** represents a logical entity and can have two values: **true** and **false**
- You can use the **Boolean()** function to find out if an expression (or a variable) is true:

```
let isLetterAndDigit = false;

console.log(Boolean(10 > 9)); // Returns true
if (5 * 'asd') {
    console.log("Truthy");
} else {
    console.log("Falsy");
}
```

Функцията **Boolean** свежда всеки тип данни до **true** или **false**

```
console.log(Boolean(100)); // Returns true
console.log(Boolean('Ami sega')); // Returns true
console.log(Boolean(0)); // Returns false
```

Всяко нещо, което има стойност е **true**, а всяко нещо, което няма стойност е **false**.

- Everything **with** a "value" is **true** (**Truthy values**)

```
let number = 1;
if (number) {
    console.log(number) // 1
}
```

Празен масив и празен обект и Boolean обаче са **true** – тъй като са референтен тип данни

```
let x = [];
console.log(Boolean(x)); // true
```

```
let xyz = {};
console.log(Boolean(xyz)); // true
```

```
let xyz = Boolean;
console.log(Boolean(xyz)); // true
```

- Everything **without** a "value" is **false** (**Falsy values**)

```
let number; // undefined
if (number) {
    console.log(number)
```

```
} else {
  console.log('false') // false
}

let x = 0;
Boolean(x); // false

let x = -0;
Boolean(x); // false

let x = '';
Boolean(x); // false

let x = false;
Boolean(x); // false

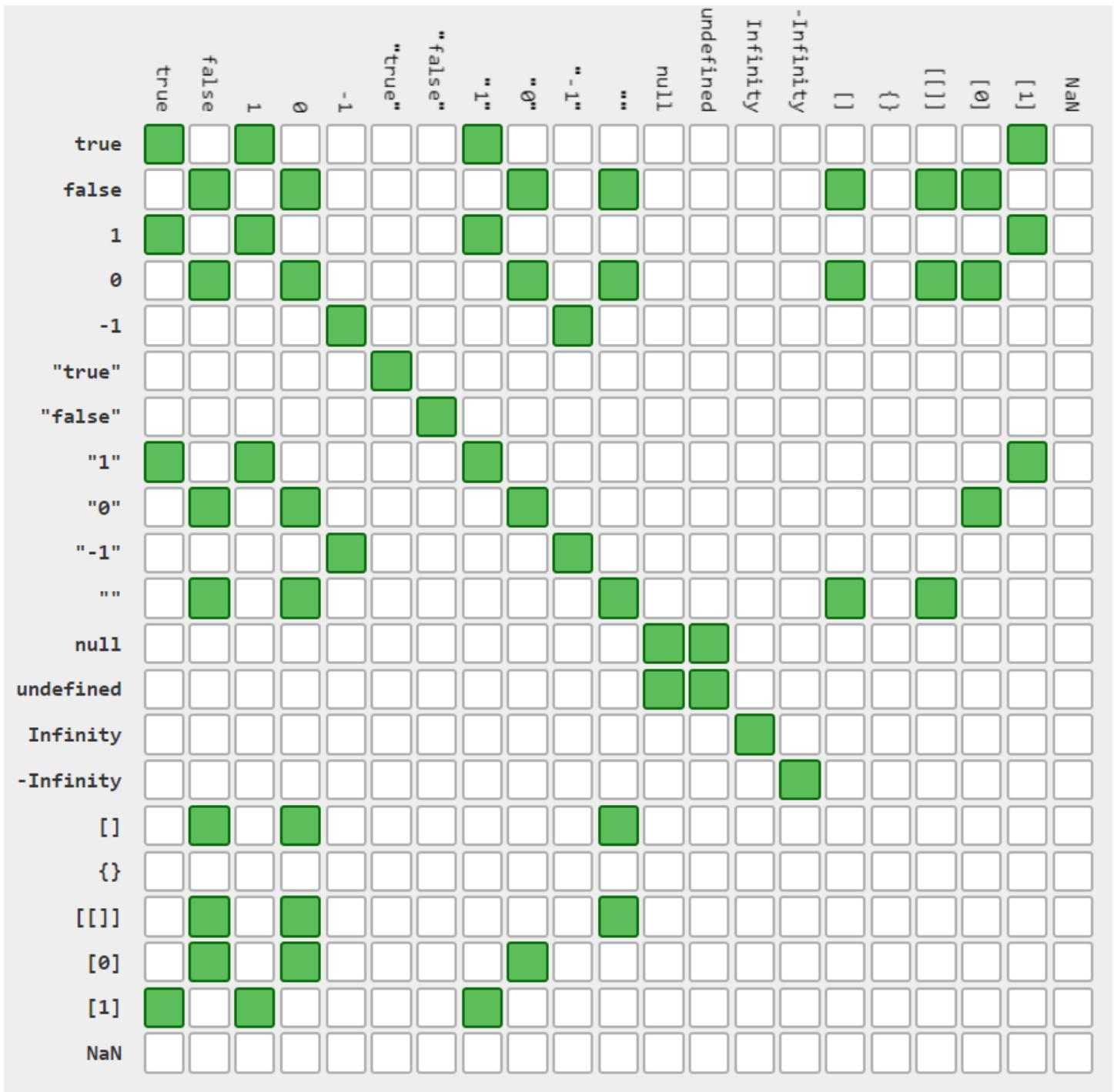
let x = null;
Boolean(x); // false

let x = 10 / 'p'; // равно на NaN, което също е false
Boolean(x); // false
```

Type coercion in JS

Таблица с двойно сравнение

`==` (negated: `!=`)



Blocked type coercion when using === (negated: !==)

When using three equals signs for JavaScript equality testing, everything is **as is**. Nothing gets converted before being evaluated.

2.7. Arrays & Objects - Reference Types

- **Arrays** are used to store multiple values in a single variable (In square brackets, separated by commas)

```
let cars = ["Saab", "Volvo", "BMW"];
```

- **Objects** containers for named values called properties or methods (In curly braces, properties are written as name: value pairs, separated by commas)

```
let person = {  
    firstName: "John",  
    lastName: "Doe",  
    age: 50,  
    eyeColor: "blue"
```

```
};
```

2.8. Typeof Operator - Checking for a Type

- Used to find the **type of a variable**
- Returns the **type** of a variable or an expression:

typeof и интервал

```
console.log(typeof "")           // Returns "string"
console.log(typeof "John")      // Returns "string"
console.log(typeof "John Doe")  // Returns "string"
console.log(typeof 0)           // Returns "number"
```

```
let result;
if (typeof a === 'string' && typeof b === 'string' && typeof c === 'string') {
    result = `${a}${b}${c}`;
}
}
```

typeof като функция

```
let n = 5;
console.log(typeof(n)); //number

if (typeof(n) === 'number') {
    console.log(n); // 5
}

if (typeof(cityName) === 'string' && typeof(population) === 'number'
&& typeof(area) === 'number')
```

Капан

```
console.log(typeof NaN); // "number"
```

2.9. Undefined and Null - Non-Existent and Empty

Undefined

- A variable without a value, has the value **undefined**. The **typeof** is also **undefined**

```
let car; // Value is undefined, type is undefined
```

- A variable can be emptied, by setting the value to **undefined**. The type will also be **undefined**

```
let car = undefined; // Value is undefined, type is undefined
```

Null

- **Null** is "**nothing**". It is supposed to be something that doesn't exist. The **typeof** null is an **object**

```
let person = {
    firstName:"John",
    lastName:"Doe",
```

```
age:50
};

person = null;
console.log(person);           // null
console.log(typeof(person)); // object
```

Null and Undefined

- **Null** is an assigned value. It means nothing
- **Undefined** typically means a variable has been declared but not defined yet
- **Null** and **Undefined** are **falsy** values
- **Undefined** and **Null** are equal in value but different in type:

```
null !== undefined // true – като стойност съвпадат, но не съвпадат като тип
null == undefined // true – като стойност съвпадат
```

Z.Z. Други действия със стринг/масив

Сплитваме по интервал и получаваме масив, обръщаме на обратно елементите му, а с `join('')` го правим на стринг

```
let password = username.split(' ').reverse().join('');
```

3. Arrays in JS

In programming **array** is a **sequence of elements**

- We can store **multiple values** in one variable
- Elements are numbered from **0** to **length-1**
- Arrays have **variable size (Array.length)** **can be resized** (unlike C# / Java)

Creating Arrays

- **Creating** an array:

```
// Array holding numbers
let numbers = [1, 2, 3, 4, 5];
```

```
// Array holding strings
```

```
let days = [
  "Monday",
  "Tuesday",
  "Wednesday",
  "Thursday",
  "Friday",
  "Saturday",
  "Sunday"
];
```

```

// Array without data
let names = [];

// Array holding mixed data
let mixedArr =
  [20, new Date(), 'hello', {x:5, y:8}];

// Creating Array with const
const arr = [1,2,3];
arr[0] = 5; //можем да сменяме елемента
arr = [4, 5 ,6]; // не можем да сменяме референцията

```

- **Accessing** array elements by index:

```
console.log(numbers[0]); // 1
```

- **Assigning values** to the array elements:

```

numbers[3] = numbers[1] + numbers[2];
console.log(numbers.length); // 5
console.log(numbers); // [1, 2, 3, 5, 5]

```

Създаване на референция от същия масив

```

let numbers = [1, 2, 3];
let numbersCopy = numbers;
numbersCopy[0] = 10;

console.log(numbers); // [10, 2, 3]
console.log(numbersCopy); // [10, 2, 3]

```

Operations

- **Replace** an element value:

```

let arr = [10, 20, 30];
arr[0] = 5; // Elements can be modified
console.log(arr); // [5, 20, 30]

```

- Check if the array **contains** the specified element:

```

console.log(arr.includes(20)); // true
console.log(arr.includes(0)); // false

```

- Finds the first/last found index for the value

```

let numbers = [1, 2, 2, 2, 5, 6, 7, 8];
let indexOfFirstFoundValue = numbers.indexOf(2);
let indexOfLastFoundValue = numbers.lastIndexOf(2);

```

```
console.log(indexOfFirstFoundValue); //1
console.log(indexOfLastFoundValue); //3
```

- Преузмеряване/**resize** на масива **and Invalid Positions**

```
let nums = [10, 20, 30];
nums[4] = 50; // Will resize the array
console.log(nums); // [10, 20, 30, <empty>, 50]
console.log(nums.length); // 5
console.log(nums[3]); // undefined

console.log(nums[-5]); // undefined
nums[-5] = -5; // Will not resize the array – слага го като ключ и е достъпен само по индекс -5, но когато го обхождаме, не е достъпен
console.log(nums[-5], nums.length); // -5 5

nums['ami sega'] = 'se taq'; // Will not resize the array
console.log(nums); // (3) [10, 20, 30, ami sega: 'se taq']
```

- Another way to **add** elements in a JS array is to use **push**:

```
let arr = [10, 20, 30];
arr.push(40); // Adds an element at the end
console.log(arr); // [10, 20, 30, 40]
```

Array Iteration

Using a normal for Loop

- To print all array elements, a for-loop can be used

```
let capitals = ['Sofia', 'Washington', 'London'];
for (let i = 0; i < capitals.length; i++){
  console.log(capitals[i]);
}
// Sofia
// Washington
// London
```

Printing with **toString()** method

- Print array elements using **toString()**

```
console.log(capitals.toString())
// Sofia, Washington, London
```

Printing Arrays with for / Join - Alternative Way to Iterate

- Use **join(separator)**:

```
let nums = [1, 2, 3];
console.log(nums.join(', ')); // 1, 2, 3
```

```
let words = [ "one", "two" ];
console.log(words.join(' - ')); // one - two
```

Можем и така да разпечатим:

```
`[${integers.join(", ")}]`
```

forEach function/method

Важно са forEach – не можем да го break-нем – той си обхожда всичко и не връща структура от данни

```
function solve(data = []) {
  let resultArr = [];
  data.forEach((currentNumber) => {
    console.log(currentNumber);
  });

  console.log(resultArr);
}

solve([5, 15, 23, 56, 35]);
```

```
// Arrow function

forEach((element) => { ... } )

forEach((element, index) => { ... } )

forEach((element, index, array) => { ... } )
```

For-in, For-of Loops – alternative way to iterate an array – only for reading the array

For-of Loop – *elements* - only for reading the array

- Iterates through all elements in a collection
- Cannot access the current index

```
let numbers = [ 1, 2, 3, 4, 5 ];
let output = '';
for (let number of numbers)
  output += `${number}`;

console.log(output);
```



```
1 2 3 4 5
```

For-in Loop – *indexes* – да не използваме for-in за масиви - only for reading the array

- Iterates through all indexes in a collection

```
let numbers = [ 5, 4, 3, 2, 1 ];
let output = '';
for (let index in numbers)
  output += `${index} `;

console.log(output);
```



Imperative versus Functional/Declarative style - .reduce() и .map()

Imperative style – обикновени цикли

Functional/Declarative style – не се интересуваме от самото изпълнение, а от логиката – .stream – този стил е много близък до многонишковото програмиране и е higher level of abstraction

Пример 1:

```
function solve(data = []) {
  let resultArr = [];
  let originalSum = 0;
  let resultSum = 0;

  //това с .map работи като цикъл и връща структура от данни
  data.map((number, index) => {
    number % 2 === 0 ? number += index : number -= index;
    resultArr.push(number);
  });

  console.log(resultArr);
  console.log(originalSum = data.reduce((sum, b) => {
    sum += b;
    return sum;
  }, 0));

  // initialValue
  console.log(resultSum = resultArr.reduce((a, b) => a + b, 0)); //starting value of a is 0
}

solve([5, 15, 23, 56, 35]);
```

4. Functions

4.1. Functions in JS

- A **function** is a **subprogram** designed to perform a particular task
 - Functions are executed when they are called. This is known as **invoking** a function
 - Values can be **passed** into functions and used within the function
 - В JS няма нужда да декларираме връщания тип данни на функцията

the function

Use camel-case

Parameter

```
function printStars(count) {  
    console.log("*".repeat(count));  
}
```

Why Use Functions?

More **manageable programming**

- **Splits** large problems into small pieces
- Better **organization** of the program
- Improves code **readability** and **understandability**

Avoiding **repeating code**

- Improves code maintainability

Code **reusability**

- Using existing functions several times

```
function solve(input) {  
    console.log(input); // Pesho  
}  
  
let functionResult = solve('Pesho');  
console.log(functionResult); // undefined
```

4.2. Declaring and Invoking Functions

Declaring Function

- Functions can have **several parameters**
- Functions **always** return a value (custom or default)

Name

Parameters

Body

```
function printText(text){  
    console.log(text);  
}
```

- Functions can be declared in two ways:

- **Function declaration = statement** - функциите имат глобален scope при function declaration

```
function printText(text) {
    console.log(text);
}
```

- **Function expression – по-странино спрямо C# и JAVA = expression** (самата функция е expression)

Променлива, на която задаваме функция = анонимна функция без име, като expression

```
var printText = function (text) {
    console.log(text);
}; //тук се слага точка и запетая
```

```
var printText = function (firstDigit, secondDigit) {
    return firstDigit + secondDigit;
};

console.log(printText(5, 6));
```

- **Function hoisting** – JS **интерпретатора** (чете отгоре надолу) хвърля един бърз поглед на целия код, и така може ако имаме **function declaration** разместени, то да ги пренареди правилно щото да се изпълни кода.

```
let functionResult = solve('Pesho'); //след това, тази част
console.log(functionResult); // се изпълнява

function solve(input) { //първо това се изпълнява/чете
    console.log(input);
}
```

Когато използваме **function expression**, то трябва да го използваме след реда, на който е декларирано. Т.е. при **function expression** няма function hoisting!!! Особено ако функцията дава резултат const.

Ако използваме var и **function expression**
или без да използваме (var, let, const),
то тогава ще се hoist-не.

Invoking a Function

- Functions are first **declared**, then **invoked** (many times)
- Functions are invoked by their name
 - **Function Declaration**

```
function printHeader() {
    console.log("-----");
}
```

- Functions can be **invoked (called)** by their name
 - **Function Invocation**

```
function main() {
```

```
    printHeader();
}
```

- A **function** can be invoked from:

- Other functions

```
function printHeader() {
    printHeaderTop();
    printHeaderBottom();
}
```

- Itself (**recursion**)

```
function crash() {
    crash();
}
```

4.3. Block scope – променливи с едно и също име

```
{
    let name = 'Pesho';
    {
        let name = 'Gosho'
        console.log(name); //Gosho
    }

    console.log(name); //Pesho
}
```

4.4. Arrow Functions

След function declaration и function expression, можем да използваме и Arrow function (вид lambda функция)

- These are functions with their own special syntax
- They accept a fixed number of arguments
- They operate in the **context** of their **enclosing scope / their this scope**

"=>" (arrow)

Функция Arrow – по-кратък вариант от function expression

```
let increment = x => x + 1;
console.log(increment(5)); // 6
```

Функция Arrow – най-кратък вариант

```
function solve(num, operation) {
    let result = operation(num);
    return result;
}
console.log(solve(5, x => x + 1)); // 6
```

Чиста функция expression

```
let increment = function (x) {
    return x + 1;
}
```

Функция с 2 параметъра

```
let sum = (a, b) => a + b;
console.log(sum(5, 6)); // 11
```

```
const multiply = (a, b) => a * b;
const divide = (a, b) => a / b;
const add = (a, b) => a + b;
const subtract = (a, b) => a - b;
```

4.5. Nested Functions

- Functions in can be **nested**, i.e. hold other functions
- Inner functions have access to variables from their parent

4.6. Value vs. Reference Types - Memory Stack and Heap

Reference vs. Value Types

- JavaScript has 5 data types that are copied by **value**:
 - **Boolean, String, Number, null, undefined**
 - These are **primitive types**
- JavaScript has 3 data types that are copied by having their **reference** copied:
 - **Array, Objects and Functions**
 - These are all technically Objects, so we'll refer to them collectively as Objects

Value Types – стоят в Stack-а

- If a primitive type is assigned to a variable, we can think of that variable as **containing** the primitive value

```
let a = 10;           let c = a;
let b = 'abc';        let d = b;
```

- They are **copied by value**

```
console.log(a, b, c, d);
// a = 10  b = 'abc'  c = 10  d = 'abc'
```

Пример - ВАЖНО:

```
function incNumber(number) {
    ++number;
    console.log("From function = " + number);
}
```

```
let number = 5;
incNumber(number); //From function = 6
console.log('outside the function ' + number); //outside the function 5
```

Reference Types – стоят в Heap-a

- Variables that are assigned a non-primitive value are given a **reference** to that value
 - That reference points to a location in memory
 - Variables don't actually contain the value but lead to the location

```
let arr = [];
let arrCopy = arr;
```

Пример – ВАЖНО – и на двете места се променя!!!

```
function incNumber(numbers) {
    ++numbers[0];
    console.log("From function = " + numbers);
}

let numbers = [1, 2, 3, 4, 5];
incNumber(numbers); //From function = 2,2,3,4,5
console.log('outside the function = ' + numbers); //outside the function = 2,2,3,4,5
```

4.7. Naming and Best Practices

Naming Functions

- Use **meaningful** names
- Should be in **camelCase**
- Names should answer the question:
 - **What does this function do?**
- If you cannot find a good name for a function, think about whether it has a **clear intent**

Naming Function Parameters

Function parameters names

- Preferred form: **[Noun]** or **[Adjective] + [Noun]**
- Should be in **camelCase**
- Should be **meaningful**
- Unit of measure should be obvious

- Each **function** should perform a **single**, well-defined task
 - A name should **describe that task** in a clear and non-ambiguous way
- **Avoid** functions **longer than one screen**
 - **Split them** to several shorter functions

```
function printReceipt(){
    printHeader();
    printBody();
    printFooter();
```

```
}
```

Code Structure and Code Formatting

- Make sure to use correct **indentation**
- Leave a **blank line** between functions, blocks, after **loops** and after **if** statements
- Always use **curly brackets** for loops and if statements bodies
- **Avoid long lines** and **complex expressions**

5. Recursion in JS

```
function factDivision(numOne, numTwo) {  
    function getFactorial(num) {  
        if (num === 1) {  
            return 1;  
        }  
        let res = num * getFactorial(num - 1);  
        return res;  
    }  
  
    let factorialOne = getFactorial(numOne);  
    let factorialTwo = getFactorial(numTwo);  
  
    console.log((factorialOne / factorialTwo).toFixed(2));  
}  
  
factDivision(2, 3);
```

6. Arrays Advanced

Advanced functionality of the array consists of the following **functions**:

push() – add to the end

- The **push** method adds **one** or **more** elements to the end of an array and returns the new length

```
let fruits = ["apple", "banana", "kiwi"];  
fruits.push("pineapple");  
console.log(fruits); // ["apple", "banana", "kiwi", "pineapple"]
```

pop() – remove from the end

- The **pop** method removes the **last** element from an array and **returns** that value to the caller
- If you call **pop()** on an empty array, it returns **undefined**

```
let myArray = ["one", "two", "three", "four", "five"];  
let popped = myArray.pop();
```

```
console.log(myArray); //["one","two","three","four"]
console.log(popped); //"five"
```

unshift() – adds elements to the beginning

```
let myArray = ["red", "green", "blue"];
myArray.unshift("purple"); // ["purple", "red", "green", "blue"]
```

shift() – remove from the beginning

```
let myArray = ["one", "two", "three", "four", "five"];
myArray.shift(); // ["two", "three", "four", "five"]
```

Обхождане на масив с **.shift()**

Има и такава опция за обхождане на масив:

```
function workShop (input) {
    let numA = input.shift(); - премахва първия елемент от масива и го връща
    let numB = input.shift();
    let name = input.shift();
}
workShop(['20', '5678', 'Ivan Ivanov']);
```

Deleting Elements

- Using **delete** changes the **element value** to **undefined** – реално не можем да изтрием веднага среден елемент на даден масив – трябва да пренареждаме елементи за да можем да изтрием краен елемент на масива.
- Use **pop()** or **shift()** instead

```
let myArray = ["one", "two", "three", "four"];
delete myArray[0];
// Changes the first element to undefined = empty
```

В JS има два типа методи на масивите – едните са **мутиращи**, другите са **немутиращи**
Мутиращи – работи директно върху масива и го променя масива

slice() – remove a range of elements

- The **slice()** function returns a newly created array
- Gets a range of elements from selected **start** (inclusive) to **end**(exclusive)
- Note that the original array will **not be modified** - **Немутиращ метод/функция**
- **It creates a new reference in the memory**

```
let myArray = ["one", "two", "three", "four", "five"];
let sliced = myArray.slice(2); //индекс 2
console.log(myArray); //["one", "two", "three", "four", "five"]
console.log(sliced); // ["three", "four", "five"]
```

```
console.log(myArray.slice(2,4)); // ["three","four"] //от индекс 2 вкл до индекс 4-1 = 3
```

Създаване копие на масив

```
let copyArray = array.slice(); - връща копие на масив/ на част от масива – копието е нова/  
друга референция в паметта  
let copyArray = array.slice(0); //същото
```

splice()

insert at position/ delete from position

- The **splice()** adds/removes items to/from an array, and **returns** the removed item(s)
- This function **changes the original array** - **Мутиращ метод/функция**

```
let nums = [5, 10, 15, 20, 25, 30];  
let mid = nums.splice(2, 3); // start, delete-count-of-elements  
console.log(mid.join('|')); // 15|20|25  
console.log(nums.join('|')); // 5|10|30
```

deletes elements of the array and inserts new elements

```
nums.splice(3, 2, "twenty", "twenty-five");  
console.log(nums.join('|')); // 5|10|15|twenty|twenty-five|30
```

```
let numbers = [1, 2, 3, 4, 5, 6, 7, 8, 9];  
let splicedElements = numbers.splice(4, 1, 50, 51); //изтрий елемент на индекс 4, само 1, и  
добави елементи 51 и 52 след това, което сме изтрили.  
console.log(splicedElements); // [5] – изрязаната част само  
console.log(numbers); // [1, 2, 3, 4, 50, 51, 6, 7, 8, 9]
```

Filtering and Transforming Elements

- .map при масиви – като цикъл с елемент и индекс

The **map()** method **creates a new array** populated with the results of calling a provided function on every element in the calling array. – връща структура от данни

Примери с .join и .map и .filter:

```
let nums = ['one', 'two', 'three', 'four'];  
console.log(nums.join('|')); // one|two|three|four
```

```
let filteredNums = nums.filter(x => x.startsWith('t'));  
console.log(filteredNums.join('|')); // two|three
```

```
let lengths = nums.map(x => x.length);  
console.log(lengths.join('|')); // 3|3|5|4
```

```
let lengths = nums.map(x => [x.length, x[0]]);
```

```
console.log(lengths.join(' | ')); // 3,o|3,t|5,t|4,f
```

Пример 1 – използване на Arrow функция:

```
function solve(arrayOfNums = []) {
    let output = arrayOfNums

    .map(Number)
    .map(x => x + 2)
    .filter(z => z > 8)
    .join(' ');

    console.log(output, typeof output, '3d argument');
}

solve(['1', '2', '3', '4']);
```

Пример 2 - използване на Arrow функция:

```
function solve(arrayOne = [], arrayTwo = []) {
    let resultArray = [];
    arrayOne.map((element, index) => { //индексът започва от 0
        index % 2 === 0 ? resultArray.push(Number(element) + Number(arrayTwo[index])) :
            resultArray.push(element + ' ' + arrayTwo[index]);
    });

    console.log(resultArray.join(' - '));
}

solve([
    ['5', '15', '23', '56', '35'],
    ['17', '22', '87', '36', '11']
]);
```

Пример 3 – използване на обикновена функция:

```
let numbers = [1, 2, 3, 4, 5, 6, 7, 8];
let oddNumbers = numbers.filter(checkOdd); //подава функцията директно във филтър

//функция тип predicate – явна, неанонимна
function checkOdd(num) {
    if (num % 2 !== 0) {
        return true;
    } else {
        return false;
    }
}

let oddNumbers = numbers.filter(function checkOdd(num) { // явна, неанонимна
    if (num % 2 !== 0) {
```

```

        return true;
    } else {
        return false;
    }
});

let oddNumbers = numbers.filter(function(num) { //функция неявна, анонимна
    if (num % 2 !== 0) {
        return true;
    } else {
        return false;
    }
});

console.log(oddNumbers); // [1, 3, 5, 7]

```

Rest operator

```

function result(...elements) { //взема ги като отделни аргументи
    for (let i = 0; i < elements.length; i++) {
        console.log(elements[i]);
    }
}

result('Ivan', 'Deyan', 'Petkan');

```

Spread operator

```

function solve(...elements) {
    console.log(elements);
}
solve(...[1, 2, 3, 4, 5]); // Прави го от колекция/масив на набор от изброени елементи.

```

Sorting Arrays

- The **sort()** function sorts the items of an array
- **It affects the original array**
- The sort order can be either **alphabetic** or **numeric**, and either **ascending (up)** or **descending (down)**
- **By default**, the **sort()** function **sorts the values as strings** in **alphabetical** and **ascending order by ASCII codes** (**малко буква а и голяма буква А не са една до друга**)
- The **sort()** function will produce an **incorrect** result when sorting numbers. You can fix this by providing a **compare function**
- **Винаги да слагаме функция в sort**

```

let nums = [20, 40, 10, 30, 100, 5];
console.log(nums.join('|')); // 20|40|10|30|100|5

```

```

nums.sort(); // Works incorrectly on arrays of numbers !!!
console.log(nums.join('|')); // 10|100|20|30|40|5

nums.sort((a, b) => a-b); // Compare elements as numbers - ascending
nums.sort((a, b) => b-a); // Compare elements as numbers - descending

console.log(nums.join('|')); // 5|10|20|30|40|100

```

Sorting String Arrays

- The **localeCompare()** method is used to compare any two characters without regard for the case used
 - It's a string method so it can't be used directly on an array
 - Pass localeCompare() as the comparison function:

```

let words = ['nest', 'Eggs', 'bite', 'Grip', 'jAw'];
words.sort((a, b) => a.localeCompare(b));
// ['bite', 'Eggs', 'Grip', 'jAw', 'nest']

```

Sorting arrays by 2 / more criteria

```

function solve(input = []) {
  let output = input.slice().sort((a, b) => {
    if (a.length === b.length) {
      return a.localeCompare(b);
    } else {
      return a.length - b.length;
    }
  });
  console.log(output.join("\n")); //печата всеки елемент на нов ред
}

solve(["alpha", "beta", "gamma"]);

```

Хитрина на JS

```

//sorting the best department
  this.departments[currentBest.name].sort((a, b) => {
    let result = b.salary - a.salary;
    if (result == 0) {
      result = a.name.localeCompare(b.name);
    }

    return result;
    return b.salary - a.salary || a.name.localeCompare(b.name); //прави същото

  })

```

7. Objects and Classes

В JS се фокусираме върху обектите, не върху класовете!!!

7.1. What Are Objects ?

- **Collection** of related data or functionality
- Consists of several variables and functions called **properties** and **methods**
- In JavaScript, at **run time** you can **add** and **remove** properties of any object



Object Definition

- We can create an object with an **object literal**, using the following syntax:

```
let person = { key1: value1, key2: value2 };
let person = {name: 'Peter', age: 20, hairColor: 'black'};
console.log(person); // {name: 'Peter', age: 20, hairColor: 'black'}
console.log(person.name); //Peter

let propertyName = 'sss';
console.log(person[propertyName]);
```

- We can define empty object and add dynamically the properties later

```
let person = {};
person.name = 'Peter';
person["lastName"] = 'Parker'; //пропърти lastName
person.age = 20;
person.hairColor = 'black';

console.log(person.lastName);

delete person.age; //изтрива пропъртито age на person
```

Object Methods

- Functions within a JavaScript object are called **methods**
- We can **define** methods using several syntaxes:

Вариант 1

```
let person = {
  sayHello: function () {
    console.log('Hi, guys');
  }
}
```

Вариант 2

```
let person = {
    sayHello() {
        console.log('Hi, guys');
    }
}
```

Вариант 3 – статично задаване на метод

```
let person = {
    name:'Peter',
    age: 20,
    hairColor: 'black',
    sing: () => console.log('lallaala'),
};

person.sing();
```

Вариант 4 – динамично задаване на метод към даден обект

```
let person = {
    name:'Peter',
    age: 20,
    hairColor: 'black',
    sing: () => console.log('lallaala')
};

person.shout = () => console.log('I shout lalalala');
person.shout();
```

- We can **add** a method to an already defined object

```
let person = { name:'Peter', age: 20 };
person.sayHello = () => console.log('Hi, guys');
```

The Object Methods

Methods:

- **Object.entries()** - returns array of all properties and their values of an object
- **Object.keys()** - returns array with all the properties
- **Object.values()** - returns array with all the values of the properties

```
let person = {
    name: 'Peter',
    age: 20,
    hairColor: 'black',
    sing: () => console.log('lallaala'),
};

console.log(Object.keys(person)); // ['name', 'age', 'hairColor', 'sing']
console.log(Object.values(person)); // ['Peter', 20, 'black', f]
```

```
let personEntries = Object.entries(person); - прави го от обект на масив  
console.log(personEntries);  
console.log(Object.fromEntries(personEntries)); - прави го от масив на обект
```

Iterate Through Keys

- Use **for-in** loop to iterate over the object properties by key:

```
let obj = { name: 'Peter', age: '18', grade: '5.50' };  
for (let key in obj) {  
    console.log(`#${key}: ${obj[key]}`); //obj[key] реално е value-то  
}  
  
let person = {  
    name: 'Peter',  
    age: 20,  
    hairColor: 'black',  
};  
  
for (let key in person) {  
    console.log(key);  
    console.log(person[key]);  
}
```

Проверка дали има пропърти

```
for (const term in sortedDictionary) {  
    if (sortedDictionary.hasOwnProperty(term)) {  
        const definition = sortedDictionary[term];  
        console.log(`Term: ${term} => Definition: ${definition}`);  
    }  
}
```

Или по този начин

```
if (catalogue[product])
```

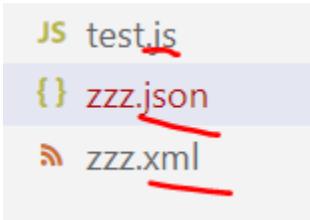
7.2. JSON - JavaScript Object Notation

- **JSON** stands for **JavaScript Object Notation** – текстов формат за данни, как да прехвърлим данни чрез текст
- **Open-standard** file format that uses text to transmit data objects
- JSON is **language independent**
- JSON is "**self-describing**" and easy to understand

JSON Usage

- Exchange data between **browser** and **server**
- JSON is a **lightweight** format compared to XML
- JavaScript has built in functions to **parse JSON** so it's easy to use

- JSON uses **human-readable** text to transmit data



Пример за JSON код

```
{
  "name": "Ivan",
  "age": 25,
  "grades": {
    "Math": [2.50, 3.50],
    "Chemistry": [4.50]
  }
}
```

Brackets define a JSON

Keys are in double quotes

Keys and values separated by :

```
{
  "name": "Ivan",
  "age": 25,
  "grades": {
    "Math": [2.50, 3.50],
    "Chemistry": [4.50]
  }
}
```

It is possible to have nested objects

In JSON we can have arrays

<https://caniuse.com/> - дали се поддържа от някой браузър даден метод

JSON Methods in JS

- We can convert object into **JSON** string using **JSON.stringify(object)** method

```
let text = JSON.stringify(obj);
```

```
let personJS = {
  name : "Stamat",
  age : 22
};
```

```
let personJSON = JSON.stringify(personJS);
console.log(personJSON);
```

- We can convert JSON string **into object** using **JSON.parse(text)** method

```
let obj = JSON.parse(text);

let personJSON = `{
  "name": "Ivan",
  "age": 25,
  "height": 190
}`;

let personObject = JSON.parse(personJSON);
console.log(personObject.name);
```

- We can convert JSON string **into array** using **JSON.parse(text)** method

```
const dataJSON = `[
  {"x": "1", "y": "2", "z": "10"},
  {"x": "7", "y": "7", "z": "10"},
  {"x": "5", "y": "2", "z": "10"}
]`;

JSON.parse(input);
[
  { x: '1', y: '2', z: '10' },
  { x: '7', y: '7', z: '10' },
  { x: '5', y: '2', z: '10' }
]
```

7.3. Classes

Функционалност, която създава обекти

- Extensible program-code-template for creating objects
- Provides **initial values** for the state of an object
- An object created by the class pattern is called an **instance** of that class
- A class has a **constructor** - subroutine called to create an object
 - It prepares the new object for use

Класът създава обекти.

Инстанция на класа е обект.

Class Declaration

```
class Student {
  constructor(name) {
    this.name = name;
  }
}
```

```

class Student {
    school = 'The Best School'; //предефинирано публично пропърти за всички инстанции на
    класа
    constructor(name, grade) {
        this.name = name;
        this.grade = grade;
    }
}

```

- Creating an **instance** of the class:

```
let student = new Student('Peter', 5.50);
```

Functions in a Class

- Classes can also have functions as property, called **methods**:

```

class Dog {
    constructor() {
        this.speak = () => {
            console.log('Woof');
        }
    }
}

```

```
let dog = new Dog();
dog.speak(); // Woof // We access the method as a regular property
```

Пример:

```

class Human {
    constructor(firstName, age){
        this.name = firstName;
        this.age = age;
    }

    sing() {
        console.log(` ${this.name} - lqlqlql`);
    }
}

let pers1 = new Human('Ivan', 28);
let pers2 = new Human('Gosho', 30);
pers1.sing(); // Ivan - lqlqlql
pers2.sing(); // Gosho - lqlqlql

```

Примерна задача за клас:

```

class Vehicle {
    constructor(type, model, parts, fuel){

```

```

    this.type = type;
    this.model = model;

    // let quality = parts.engine * parts.power;
    // this.parts = {
    //     engine : parts.engine,
    //     power : parts.power,
    //     quality : quality,
    // }

    this.parts = parts;
    this.parts.quality = this.parts.engine * this.parts.power;

    this.fuel = fuel;
}

drive(fuelLoss){
    this.fuel -= fuelLoss;
}
}

let parts = { engine: 6, power: 100 };
let vehicle = new Vehicle('a', 'b', parts, 200);
vehicle.drive(100);
console.log(vehicle.fuel); //100
console.log(vehicle.parts.quality); //600

```

8. Associative Arrays

- Arrays indexed by **string keys**
- Hold a set of pairs **[key => value]**
 - The key can either be an **integer** or a **string**
 - The **value** can be of **any** type

8.1. Associative Arrays

Declaration

- An associative array in JavaScript is just an **object**
- We can declare it **dynamically**

```

let assocArr = {
    'one': 1,
    'two': 2,
    'three': 3
};

```

Attributes

- The syntax for **accessing** the value of a key is

```

assocArr['key'] // person['age']
assocArr[key] // key = "age"; person[key]

```

- **Assigning** a value to a variable

```
let age = assocArr[key];
```

Using for – in

- We can use **for-in** loop to iterate through the keys

```
let assocArr = {};
assocArr['one'] = 1;
assocArr['two'] = 2;
assocArr['three'] = 3;
```

```
for (let key in assocArr) {
  console.log(key + " = " + assocArr[key]);
}
one = 1
two = 2
three = 3
```

Using ForEach

- We can also use **forEach** loop to iterate through the **keys**

```
let assocArr = {};
assocArr['one'] = 1;
assocArr['two'] = 2;
assocArr['three'] = 3;
```

```
Object.keys(assocArr)
.forEach((i) => {
  console.log(` ${i} = ${assocArr[i]}`)
});
```

- We can also use **forEach** loop to iterate through the **entries**

```
Object.entries(assocArr)
.forEach((kvp) => {
  console.log(` ${kvp[0]} = ${kvp[1]}`)
});
```

8.2. Maps

- A **Map** object stores its elements in **insertion order** - за разлика, при обектите реда на вкарване може да е различен!!!
- A for-of loop returns an array of **[key, value]** for each iteration
- Pure **JavaScript objects** are like **Maps** in that both let you:
 - Set **keys to values**
 - Delete keys
 - **The key can be an integer, string, Boolean, други**

Detect whether something is stored in a key

При обектите, задължително ключът е string. Докато при Map може ключът да е всякакъв тип данни

Ако искаме често да изтриваме неща, то по-добре е да използваме map отколкото обект.

Adding/Accessing Elements

- `.set(key, value)` - adds a new key-value pair

```
let map = new Map();
map.set(1, "one"); // key - 1, value - one
map.set(2, "two"); // key - 2, value - two
```

```
let map = {
    2: 2, 3: 3, 4: 4, 5: 5, 6: 6, 7: 7, 8: 8, 9: 9, 10: 10,
    J: 11, Q: 12, K: 13, A: 14, S: 4, H: 3, D: 2, C: 1
};
```

Пример 1:

```
let storage = new Map();

for (const line of input) {
    let [item, quantityText] = line.split(' ');
    let quantity = Number(quantityText);
    if (storage.has(item)) {
        let currQuantity = storage.get(item);
        storage.set(item, currQuantity + quantity);
    } else {
        storage.set(item, quantity);
    }
}
```

Пример 2:

```
let neighborhoods = new Map();
input.shift()
    .split(' ')
    .forEach(el => neighborhoods.set(el, []));
```

- `.get(key)` - returns the value of the given key

```
map.get(2); // two
map.get(1); // one
```

```
let curr = neighborhoods.get(neighborhoodName);
curr.push(person);
```

- **Adding dynamically keys and values**

```
function solve(input = []) {
```

```

let phonebook = {};

for (let i = 0; i < input.length; i++) {
    let entry = input[i].split(" ");
    let name = entry[0];
    let number = entry[1];

    phonebook[name] = number;
}

for (const name in phonebook) {
    console.log(` ${name} -> ${phonebook[name]}`);
}
}

solve(['Tim 0834212554',
    'Peter 0877547887',
    'Bill 0896543112',
    'Tim 0876566344']
);

```

Contains / Delete

- `.has(key)` - checks if the map has the given key

```
map.has(2); // true
```

```
map.has(4); // false
```

- `.delete(key)` - removes a key-value pair

```
map.delete (1); // Removes 1 from the map
```

- `.clear()` - removes all key-value pairs

```
map.clear();
```

Iterators

- `.entries()` - returns Iterator - array of [key, value]
- `.keys()` - returns Iterator with all the keys
- `.values()` - returns Iterator with all the values

```
let entries = Array.from(map.entries()); // [ [2, 'two'], [3, 'three'] ]
let keys = Array.from(map.keys()); // [2, 3]
let values = Array.from(map.values()); // ['two', 'three']
```

**These methods return an `Iterator`,
transform it into an `Array`**

Iterating a Map

- To print a map simply use one of the `iterators` inside a `for-of`

```
let iterable = Array.from(phonebookMap.entries());
for (let kvp of iterable) {
    let name = kvp[0];
    let number = kvp[1];
    console.log(` ${name} -> ${number}`);
}
```

```

}

for (const [key, value] of storage) {
    console.log(` ${key} -> ${value}`);
}

```

Map printing/iterating - one more option with forEach

With forEach and key-value pair

```

let storage = new Map();
storage.forEach((value, key) => console.log(` ${key} -> ${value}`));

```

От Map през Iterator към Array:

```
let phonebookArray = Array.from(phonebookMap.entries());
```

От Map през Iterator към Object:

```
let phonebookObject = Object.fromEntries(phonebookMap.entries());
```

От Object през Iterator към Map:

```
let phonebookMap = new Map(Object.entries(phonebookObject));
```

От Object към Array:

```

let person = {
    name: 'Peter',
    age: 20,
    hairColor: 'black',
    sing: () => console.log('lallaala'),
};

let personEntries = Object.entries(person); - прави го от обект на масив

// [key, value]
Object.entries(words)
    .sort((a, b) => b[1] - a[1])
    .forEach(x => console.log(` ${x[0]} - ${x[1]}`));

```

Map Sorting

- To **sort** a Map, firstly transform it into an **array**
- Then use the **sort()** method

Sort **ascending** by value

```

let map = new Map();
map.set("one", 1);
map.set("eight", 8);
map.set("two", 2);
let sorted = Array.from(map.entries())
    .sort((a, b) => a[1] - b[1]);

```

```
for (let kvp of sorted) {  
    console.log(`${kvp[0]} -> ${kvp[1]}`);  
}
```

8.3. Sets - Storing Unique Elements

- Store **unique values** of any type, whether **primitive** values or **object** references – **by insertion order**
- Set objects are **collections** of values
- Can **iterate** through the elements of a set in **insertion** order

Пример 1:

```
let set = new Set([1, 2, 2, 4, 5]);  
// Set(4) { 1, 2, 4, 5 }  
console.log(set.has(1));  
// Expected output: true
```

Пример 2:

```
let things = new Set();  
things.add(1);  
things.add(2);  
things.add(3);  
things.add(4);  
  
console.log(things.has(4));  
  
console.log(Array.from(things)); // [1, 2, 3, 4]  
console.log(...things); // 1 2 3 4
```

Все тая дали ще използваме `.keys()` или `.values()` метода – при сета ще върне един и същи масив от данни.

Уникални value-та.

В JAVA мы называем просто keys

Methods

[Set.prototype\[@@iterator\]\(\)](#)

`Set.prototype.add()`

`Set.prototype.clear()`

`Set.prototype.delete()`

`Set.prototype.entries()`

`Set.prototype.forEach()`

`Set.prototype.has()`

`Set.prototype.values()`

9. Text Processing

9.1. Strings – what is string

Strings Are Immutable

- Strings are **immutable** (read-only) sequences of characters
- Accessible by **index**

```
let str = "Hello, JS";
let ch = str[2]; // Expected output: l
ch = str.charAt(2); // Expected output: l
// Both declarations are the same
```

9.2. Manipulating Strings

Concatenating

- Use the "+" or the "+=" operators

```
let text = "Hello" + ", ";
// Expected output: "Hello, "
text += "JS!"; // "Hello, JS!"
```

- Use the **concat()** method

```
let greet = "Hello, ";
let name = "John";
let result = greet.concat(name);
console.log(result); // Expected output: "Hello, John"
```

```
let lines = ['Hello', ' ', 'World', '!'];
let message = ''.concat(...lines);
console.log(message);
```

Searching for Substrings

- **indexOf(substr)**

```
let str = "I am JavaScript developer";
console.log(str.indexOf("Java")); // Expected output: 5
console.log(str.indexOf("java")); // Expected output: -1
```

- **lastIndexOf(substr)**

```
let str = "Intro to programming";
let last = str.lastIndexOf("o");
console.log(last); // Expected output: 11
```

- **Търсене на повече от едно съвпадение**

```
let text = 'Az sym bylgarche obicham sym';
let index = text.indexOf('sym');

while (index >= 0) {
  console.log(index); //3 и 25
```

```
    index = text.indexOf('sym', ++index); //втори аргумент – от кой индекс да започне търсенето. По подразбиране(без втори аргумент), търсенето започва от 0ия индекс.  
}
```

Extracting Substrings

- **substr(startIndex, length)** - deprecated in some Web browsers!!!

```
let str = "I am JavaScript developer";  
let sub = str.substr(5, 10);  
console.log(sub); // Expected output: JavaScript
```

- **substring(startIndex, endIndex?)**

```
let str = "I am JavaScript developer";  
let sub = str.substring(5, 10); //до 9ти индекс, т.е. до 10ти индекс exclusive  
console.log(sub); // Expected output: JavaS
```

The slice() method for string

.slice() функция при стрингове

```
let color = 'red';  
console.log(color.slice(1, color.length)); // ed
```

Replacing text

- **replace(search, replacement)** – заменя първото срещнато само

```
let text = "Hello, john@softuni.bg, you have been using john@softuni.bg in your registration.";  
let replacedText = text.replace(".bg", ".com");  
console.log(replacedText);  
// Hello, john@softuni.com, you have been using  
john@softuni.bg in your registration.
```

Ако използваме Regex, заменя всички срещания

Подменяме всички съвпадения без Regex

```
let text = 'Az sym bylgarche obicham ...bylgarche';
let replacedText;

while (text.indexOf('bylgarche') >= 0) {
    text = text.replace('bylgarche', 'evropeiche'); //само едно съвпадение
}

console.log(text);
```

Splitting and Finding

- **split(separator)**

```
let text = "I love fruits";
let words = text.split(' ');
console.log(words); // Expected output: ['I', 'love', 'fruits']
```

- **includes(substr)**

```
let text = "I love fruits.";
console.log(text.includes("fruits")); // Expected output: True
console.log(text.includes("banana")); // Expected output: False
```

Repeating Strings

- **str.repeat(count)** - Creates a new string repeated count times

```
let n = 3;
for (let i = 1; i <= n; i++) {
    console.log('*'.repeat(i));
}

// *
// **
// ***

function solve(text, word) {
    while (text.includes(word)) {
        text = text.replace(word, '*'.repeat(word.length));
    }

    return text;
}

console.log(solve("A small sentence with some words", "small"));
//A ***** sentence with some words
```

Trimming Strings

- Use **trim()** method to remove **whitespaces** (spaces, tabs, no-break space, etc.) from **both ends** of a string

```
let text = "    Annoying spaces      ";
```

```
console.log(text.trim()); // Expected output: "Annoying spaces"
```

- Use **trimStart()** or **trimEnd()** to remove whitespaces **only** at the beginning or at the end

```
let text = "    Annoying spaces      ";
text = text.trimStart();
text = text.trimEnd();
console.log(text); // Expected output: "Annoying spaces"
```

Starts With/Ends with

- Use **startsWith()** to determine whether a string **begins** with the characters of a specified substring

```
let text = "My name is John";
console.log(text.startsWith('My')); // Expected output: true
```

- Use **endsWith()** to determine whether a string **ends** with the characters of a specified substring

```
let text = "My name is John";
console.log(text.endsWith('John')); // Expected output: true
```

Padding at the Start and End – в случай, че не му достигат

- Use **padStart()** to add to the current string **another substring** at the **start** until a **length** is reached

```
let text = "010";
console.log(text.padStart(8, '0')); // Expected output: 00000010
//Общо 8 символа като допълва отпред с '0'
```

- Use **padEnd()** to add to the current string **another substring** at the **end** until a **length** is reached

```
let sentence = "He passed away";
console.log(sentence.padEnd(20, '.'));
// Expected output: He passed away.....
```

DOM operations

```
element.textContent = text.replaceAll(match, replacer); //заменя всички – работи само в браузъра, а в Judge все още не
```

```
element.textContent = text.split(match).join(replacer); //пак заменя всички
```

```
element.textContent = text.replace(match, replacer); //заменя само първото съвпадение
```

Вариант с regex и функцията replace

```
<script>
  /**
   * @param {string} element
   */
  function edit(ref, match, replacer) {
    const content = ref.textContent;
    const matcher = new RegExp(match, 'g'); //флаг global
    const edited = content.replace(matcher, replacer);
    ref.textContent = edited;
```

```
    }  
</script>
```

10. Regular Expressions

10.1. Regular Expressions - Definition and Classes

What Are Regular Expressions?

- Regular expressions (regex)
 - Match text by pattern
- Patterns are defined by special syntax, e.g.
 - **[0-9]+** matches non-empty sequence of digits
 - **[A-Z][a-z]*** matches a capital + small letters

Play with regex live at: regexp.com, regex101.com

Character Classes: Ranges

- **[n|v|j]** matches any character that is either **n**, **v** or **j**

node.js v0.12.2

- **[^abc]** – matches any character that is **not a, b or c**

Abraham

- **[0-9]** – character range: matches any digit from **0** to **9**

John is 8 years old.

Predefined Classes

- **\w** – matches any **word character** (a-z, A-Z, 0-9, _)
- **\W** – matches any **non-word character** (the opposite of \w)
- **\s** – matches any **white-space character**
- **\S** – matches any **non-white-space** character (opposite of \s)
- **\d** – matches any **decimal digit** (0-9)
- **\D** – matches any **non-decimal character** (the opposite of \d)

10.2. Quantifiers - grouping

Quantifiers

- ***** – matches the previous element **zero or more times**

\+\d* → **+359885976002 a+b**

- **+** – matches the previous element **one or more times**

\+\d+ → **+359885976002 a+b**

- **?** – matches the previous element **zero or one time**

\+\d? → +359885976002 a+b

- {3} – matches the previous element exactly 3 times

\+\d{3} → +359885976002 a+b

- {2, 4} – custom range - matches the previous element from 2 to 4 times times

Grouping Constructs

- (**subexpression**) – captures the matched subexpression as numbered group

\d{2}-(\w{3})-\d{4} → 22-Jan-2015

- (?**:subexpression**) – defines a non-capturing group

Не можем да я реферираме с backreference и не можем да вземаме елементи от тази група за други цели, но служи като част от pattern-a.

Не ги брои като поредна група като е non-capturing.

^(?:Hi|hello), \s*(\w+)\$ → Hi, Peter

Пример:

Група 1, Група 3, Група 4 и Група 5 са данните.

Група 2 е всички символи без символите (^|\$%.)*

REGULAR EXPRESSION 2 matches (198 steps, 0.9ms)

```
:/ ^%([A-Z][a-z]+)%([^\$%.]*<(\w+)>(?:\2)\||(?:\2)(\d+)|\g{2}\|(?:(\d+)\|(?:(\d+)\D*\d+\.\?\d*)\$\$|
```

TEST STRING

```
%InvalidName%<Croissant>|2|10.3$  
%Peter%<Gum>1.3$  
%Maria%<Cola>|1|2.4  
%Valid%<Valid>|10|valid20$  
%Maria%<Cola>|1|2.4$  
end of shift
```

- (?<name>**subexpression**) – defines a named capturing group

(?<day>\d{2})-(?<month>\w{3})- (?<year>\d{4}) → 22-Jan-2015

- **Nested groups** – добре е да използваме a named capturing group

В противен случай, гледаме коя скоба е първа и тя е първата група с номер 1

Start and End string pattern

^\w+@[a-zA-Z]+\.[a-zA-Z]+\$

^ - start

\$ - end

\b – разграничава дума от дума = разграничава word-character from non-word-character – служи за начало/край

10.3. Backreferences - numbered Capturing Group

Backreferences Match Previous Groups

- \number – matches the value of a numbered capture group

<(\w+)[^>]*>.*?<\1> – един или повече символи в триъгълните скоби без символа > нула или повече пъти, след това само точка е всеки символ нула или повече пъти (но само нула или една такава поредност), след това група едно се повтаря.

```
<b>Regular Expressions</b> are cool!
<p>I am a paragraph</p> ... some text after
Hello, <div>I am a<code>DIV</code></div>!
<span>Hello, I am Span</span>
<a href="https://softuni.bg/">SoftUni</a>
```

10.4. Regular Expressions in JS

Regex in JS

- In JS you construct a regular expression in one of two ways:
 - Regular Expression Literal – **ако си го пишем сами**

```
// Provides compilation when the script is loaded
```

```
let regLiteral = /[A-Za-z]+/gm
```

Буквата g обозначава, че е глобално, т.е. не намира само първото съвпадение, а намира всички съвпадения като въръща масив.

Шаблонът го слагаме в /израз/g

g – global

m - multiline

```
let result = '';
const pattern = /\((.+?)\)/g;    литерал за тип регекс
console.log(pattern instanceof RegExp); //true
```

- The constructor function **RegExp** – **ако ни идва отвън флаговете (g, m, i, s, u, y)**

```
// Provides runtime compilation
```

```
// Used when the pattern is from another source
```

```
let regExp = new RegExp('[A-Za-z]+', 'g');
```

Първите две се изпълняват върху самият обект от тип RegExp, а останалите се изпълняват върху стрингове!!!

RegExp

string

Methods that use regular expressions

Method	Description
<code>exec()</code>	Executes a search for a match in a string. It returns an array containing the results.
<code>test()</code>	Tests for a match in a string. It returns <code>true</code> or <code>false</code> .
<code>match()</code>	Returns an array containing all of the matches, including the whole match and the captured groups.
<code>matchAll()</code>	Returns an iterator containing all of the matches, including the whole match and the captured groups.
<code>search()</code>	Tests for a match in a string. It returns the index of the first match found.
<code>replace()</code>	Executes a search for a match in a string and replaces it with another string.
<code>split()</code>	Uses a regular expression or a fixed string to split a string into an array of substrings.

Using the Exec() Method

- The method `exec(string text)`
 - Works with a pointer & returns the **groups**

```
let text = 'Peter: 123 Mark: 456';
let pattern = /([A-Z][a-z]+): (\d+)/g; //като е глобален флаг, помни докъде е стигнало
let firstMatch = pattern.exec(text);
let secondMatch = pattern.exec(text);

console.log(firstMatch);
```

По резултата се ориентирам коя група търся = console.log-ваме за да разберем

```
[  
  'Peter: 123',  
  'Peter',  
  '123',  
  index: 0,  
  input: 'Peter: 123 Mark: 456',  
  groups: undefined  
]
```

```
console.log(secondMatch);  
[  
  'Mark: 456',  
  'Mark',  
  '456',  
  index: 11,  
  input: 'Peter: 123 Mark: 456',  
  groups: undefined  
]
```

```
console.log(firstMatch[0]) // Peter: 123
console.log(firstMatch[1]); // Peter
console.log(firstMatch[2]); // 123
```

Пример 1:

```
let match = pattern.exec(text);
while (match != null) {
  result += match[1];
  result+= ';';
  match = pattern.exec(text);
}

return result;
```

Пример 2:

```
const pattern = /^[a-z]+@[a-z]+\.[a-z]+$/;

if (pattern.test(target.value)) {
  target.classList.remove('error');
} else {
  target.classList.add('error');
}
```

Validating String by Pattern

- The method **test(string text)**
 - Determines whether there is a match

```
let text = 'Today is 2015-05-11';
let pattern = /\d{4}-\d{2}-\d{2}/g;

let containsValidDate = pattern.test(text);
console.log(containsValidDate); // true
```

Checking for Matches

The method match(regex)

- Returns an **array** of all matches (strings)
- `let text = 'Peter: 123 Mark: 456';`
- `let pattern = /([A-Z][a-z]+): (\d+)/g;`
- `let matches = text.match(pattern);` //връща масив от стрингове

```
console.log(matches);
[ 'Peter: 123', 'Mark: 456' ]
console.log(matches.length); // 2
console.log(matches[0]); // Peter: 123
console.log(matches[1]); // Mark: 456
```

Ако флагът не е global:

```
let text = 'Peter: 123 Mark: 456';

let pattern = /([A-Z][a-z]+): (\d+)/g;
let matches = text.match(pattern); //връща само първото съвпадение
console.log(matches);
```

```
[
  'Peter: 123',
  'Peter',
  '123',
  index: 0,
  input: 'Peter: 123 Mark: 456',
  groups: undefined
]
```

Ако флагът не е global (достъпваме само първото съвпадение) и имаме именувана група

```
let text = 'Peter: 123 Mark: 456';
let pattern = /(?(<username>[A-Z][a-z]+): (\d+)/;
let matches = text.match(pattern);
console.log(matches);
console.log(matches.index);
console.log(matches.input);
console.log(matches.groups.username);
console.log(matches.groups);
[
  'Peter: 123',
  'Peter',
  '123',
  index: 0,
  input: 'Peter: 123 Mark: 456',
  groups: [Object: null prototype] { username: 'Peter' }
]
```

```
matchAll(pattern)
function solve(input) {
  const pattern = /^>>([a-zA-Z]+)<<([0-9]+\.?[0-9]*)(\d+)$/gm;
  let lines = input
    .slice(0, input.indexOf('Purchase'))
    .join("\n");

  let arr = Array.from(lines.matchAll(pattern)); //задължително използваме Array.from
  return arr;
}

console.log(
  solve([
    ">>Sofa<<312.23!3",
    ">>TV<<300!5",
    ">Invalid<<!5",
    "Purchase"
  ]);
//хваща само 2те съвпадения
[
```

```
'>>Sofa<<312.23!3',
'Sofa',
'312.23',
'3',
index: 0,
input: '>>Sofa<<312.23!3\n>>TV<<300!5\n>Invalid<<!5',
groups: undefined
],
[
  [
    '>>TV<<300!5',
    'TV',
    '300',
    '5',
    index: 17,
    input: '>>Sofa<<312.23!3\n>>TV<<300!5\n>Invalid<<!5',
    groups: undefined
  ]
]
```

Searching with Regex

Replacing with Regex

- The method **replace(regex, string replacement)**
 - Replaces all strings that match the pattern with the provided replacement – само ако флагът е глобален

```
let text = 'Peter: 123 Mark: 456';
let replacement = '999';
let regex = /\d{3}/g;
let result = text.replace(regex, replacement);
console.log(result); // Peter: 999 Mark: 999
```

Splitting with Regex

- The method **split(regex)**
 - Splits the text by the pattern
 - Returns an array of strings

```
let text = '1 2 3      4';
let regex = /\s+/g;
let result = text.split(regex);
console.log(result) // ['1', '2', '3', '4'];
```

Grouping example 1:

```
function solve(input) {
  const pattern = /^>>([a-zA-Z]+)<<([0-9]+\.[0-9]*)!(\d+)/gm;
  let lines = input
```

```

    .slice(0, input.indexOf('Purchase'))
    .join("\n");

let arr = Array.from(lines.matchAll(pattern));
let names = `Bought furniture:`;

arr.forEach(match => {
  console.log(match);
  names += `\n` + match[1];
  cost += Number(match[2]) * Number(match[3]);
});

return names;
}

console.log(
  solve([
    ">>Sofa<<312.23!3",
    ">>TV<<300!5",
    ">Invalid<<!5",
    "Purchase"
  ])
);

[
  '>>Sofa<<312.23!3', //нулевият елемент е целият мачнат string
  'Sofa', //първи елемент от първи match
  '312.23',
  '3',
  index: 0,
  input: '>>Sofa<<312.23!3\n>>TV<<300!5\n>Invalid<<!5',
  groups: undefined
]
[
  '>>TV<<300!5', //нулевият елемент е целият мачнат string
  'TV', //първи елемент от втори match
  '300',
  '5',
  index: 17,
  input: '>>Sofa<<312.23!3\n>>TV<<300!5\n>Invalid<<!5',
  groups: undefined
]
//  

Bought furniture:  

Sofa  

TV

```

Grouping example 2:

11. Syntax, Functions and Statements

Operators, Parameters, Return Value, Arrow Functions

- JavaScript is a **dynamic programming language**
 - Operations otherwise done at **compile-time** can be done at **run-time**
- It is **possible** to change the **type** of a variable or add new properties or methods to an object **while** the program is **running**
- In **static programming languages**, such changes are normally **not possible**

11.1. Data Types & Variables

Data Types

- Seven **data types** that are **primitives**
 - **String** - used to represent textual data
 - **Number** - a numeric data type
 - **Boolean** - a logical data type
 - **Undefined** - automatically assigned to variables
 - **Null** - represents the **intentional absence** of any object value
 - **BigInt** - represent integers with **arbitrary precision**
 - **Symbol** - **unique** and **immutable** primitive value

- **Reference types – Object**

Identifiers

- An **identifier** is a sequence of characters in the code that identifies a **variable**, **function**, or **property** – **името се нарича идентификатор**
- In JavaScript, identifiers are **case-sensitive** and can contain Unicode **letters**, **\$**, **_**, and **digits** (0-9), but should **not** start with a digit

```
let _name = "John";  
  
function $sum(x, y) {  
    return x + y;  
}  
  
let 9nine = 'nine'; //SyntaxError: Unexpected number
```

Variable Values

- Used to **store** data values
- Variables that are assigned a **non-primitive** value are given a **reference** to that value
- **Undefined** - a **variable** that has been declared with a keyword, but not given a value

```
let a;  
console.log(a) //undefined  
  
• Undeclared - a variable that hasn't been declared at all  
console.log(undeclaredVariable);
```

```
//ReferenceError: undeclaredVariable is not defined
```

- **let, const** and **var** are used to declare variables
 - **let** - allows **reassignment**

```
let name = "George";  
name = "Maria";
```

- **const** - once assigned it **cannot** be modified

```
const name = "George";  
name = "Maria"; // TypeError
```

- **var** - defines a variable in the function scope **regardless** of block scope

```
var name = "George";  
name = "Maria";
```

Legacy Variable Declaration

- You will see **var** used in old examples
- Using **var** to declare variables is a **legacy** technique
- Since **ES2015 let** and **const** are available
- **var** introduces function scope **hoisting**
 - Will be discussed later in the lesson
- There is no good reason to **ever** use **var**!

Variable Scopes

- **Global scope** – Any variable that's **NOT** inside any **function** or **block** (a pair of curly braces);
- **Functional scope** – Variable declared **inside a function** is inside the local scope;
- **Block scope** – **let** and **const** declares **block** scoped variables

Dynamic Typing

- Variables in JavaScript are **not** directly **associated** with any particular **value type**
- Any variable **can** be assigned (and re-assigned) values of all types

```
let foo = 42; // foo is now a number  
foo = 'bar'; // foo is now a string  
foo = true; // foo is now a boolean
```

- **NOTE:** The use of dynamic typing is considered a bad practice!

11.2. Functions

- **Function** - named list of instructions (statements and expressions)
- Can take **parameters** and return **result**
 - Function names and parameters use **camel case**
 - The { stays at the same line

Declaring Functions

Function declaration

```
function walk() {  
    console.log("walking");  
}
```

Function expression

```
const walk = function () {  
    console.log("walking");  
}
```

Arrow functions expression

```
const walk = () => {  
    console.log("walking");  
}
```

Generators

```
//безкраен цикъл  
function* getIdGenerator(){  
    let lastId = 0;  
    while (true) {  
        lastId++;  
        yield lastId; //като return работи, но изчакай и за следващо извикване  
    }  
}
```

```
const idGenerator1 = getIdGenerator();  
const idGenerator2 = getIdGenerator();
```

```
console.log(idGenerator1.next());  
console.log(idGenerator1.next());  
console.log(idGenerator1.next());  
console.log(idGenerator2.next());  
console.log(idGenerator1.next());
```

```
{ value: 1, done: false }  
{ value: 2, done: false }  
{ value: 3, done: false }  
{ value: 1, done: false }  
{ value: 4, done: false }
```

```
const idGenerator = getIdGenerator();  
for(let id of idGenerator){  
    console.log(id);  
    if (id > 10) {  
        break;  
    }
```

```
}
```

```
1
```

```
2
```

```
3
```

```
4
```

```
5
```

```
6
```

```
7
```

```
8
```

```
9
```

```
10
```

```
11
```

Parameters and Returned Value

- You can receive parameters with **no value**
- The **unused parameters** are ignored

```
function foo(a,b,c){  
    console.log(a);  
    console.log(b);  
    console.log(c); //undefined  
}  
foo(1,2)
```

```
function foo(a,b,c){  
    console.log(a);  
    console.log(b);  
    console.log(c);  
}  
foo(1,2,3,6,7)
```

- Functions can yield a value with the **return** operator

```
function identity(param) {  
    return param;  
}  
console.log(identity(5)) // 5
```

Object Methods and Standard Library

- Any object may have **methods**
 - **Functions** that operate from the **context** of the object
 - Accessed as a **property** using the **dot-notation**

```
let myString = 'Hello, JavaScript!';  
console.log(myString.toLowerCase());  
// hello, javascript!
```

- JavaScript has a large **standard library**
 - **Math, Number, Date, RegExp, JSON** and more
 - For more information, [visit MDN](#)

Default Function Parameter Values

- Functions can have **default parameter** values

```
function printStars(count = 5) {  
    console.log("*".repeat(count));
```

```
}
```

```
printStars(); // *****
printStars(2); // **
printStars(3, 5, 8); // ***
```

11.3. Operators and Statements

Arithmetic Operators

- **Arithmetic operators** - take numerical values (either literals or variables) as their operands
 - Return a single numerical value
 - Addition (+)
 - Subtraction (-)
 - Multiplication (*)
 - Division (/)
 - Remainder (%)
 - Exponentiation (**)

```
let a = 15;
let b = 5;
let c;
c = a + b; // 20
c = a - b; // 10
c = a * b; // 75
c = a / b; // 3
c = a % b; // 0
c = a ** b; // 155 = 759375
```

Assignment Operators

- **Assignment operators** - assign a value to its left operand based on the value of the right operand

Name	Shorthand operator	Basic usage
Assignment	$x = y$	$x = y$
Addition assignment	$x += y$	$x = x + y$
Subtraction assignment	$x -= y$	$x = x - y$
Multiplication assignment	$x *= y$	$x = x * y$
Division assignment	$x /= y$	$x = x / y$
Remainder assignment	$x %= y$	$x = x \% y$
Exponentiation assignment	$x **= y$	$x = x ** y$

Comparison Operators

```
console.log(1 == '1'); // true
console.log(1 === '1'); // false
console.log(3 != '3'); // false
console.log(3 !== '3'); // true
console.log(5 < 5.5); // true
console.log(5 <= 4); // false
console.log(2 > 1.5); // true
```

```
console.log(2 >= 2);      // true
console.log(5 > 7 ? 4 : 10); // 10
```

Truthy and Falsy Values

- "truthy" - a value that **coerces** to **true** when **evaluated** in a boolean context
- The following values are "falsy" - **false**, **null**, **undefined**, **NaN**, **0**, **On** and **""**

```
function logTruthiness(val) {
  if (val) {
    console.log("Truthy!");
  } else {
    console.log("Falsy.");
  }
}

logTruthiness(3.14);      //Truthy!
logTruthiness({});        //Truthy!
logTruthiness(NaN);       //Falsy.
logTruthiness("NaN");     //Truthy!
logTruthiness([]);        //Truthy!
logTruthiness(null);      //Falsy.
logTruthiness("");         //Falsy.
logTruthiness(undefined); //Falsy.
logTruthiness(0);          //Falsy.
```

Капан

```
console.log(typeof NaN); // "number"
```

Logical Operators

- **&& (logical AND)** - **returns the leftmost "false" value or the last truthy value**; if all are true – работи по-малко по-различен начин – връща true/false/null/NaN/undefined и т.н.

```
const val = 'yes' && null && false
console.log(val); // null
const val1 = true && 5 && 'yes';
console.log(val1); // 'yes'
```

- **|| (logical OR)** - **returns the leftmost "true" value or the last falsy value**; if all are false. - работи по-малко по-различен начин – връща true/false/null/NaN/undefined и т.н.

```
const val = false || 5 || '';
console.log(val); // 5
```

```
const val1 = null || NaN || undefined;
console.log(val1); // undefined
```

- **! (logical NOT)** - Returns **false** if its single operand can be converted to **true**; otherwise, returns **true**

Typeof Operator

- The **typeof** operator returns a string indicating the type of an operand

typeof и интервал

```
const val = 5;
console.log(typeof val);    // number

const str = 'hello';
console.log(typeof str);    // string

const obj = {name: 'Maria', age:18};
console.log(typeof obj);    // object
```

typeof като функция

```
let n = 5;
console.log(typeof(n)); //number

if (typeof(n) === 'number') {
  console.log(n); // 5
}

if (typeof(cityName) === 'string' && typeof(population) === 'number'
&& typeof(area) === 'number')
```

Some interesting examples

```
console.log(typeof NaN);           //number
console.log(NaN === NaN);          //false
console.log(typeof null);          //object(legacy reasons)
console.log(new Array() == false);  //true
console.log(0.1 + 0.2);            //0.3000000000000004
console.log((0.2 * 10 + 0.1 * 10) / 10); //0.3
```

instanceof Operator

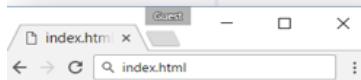
```
let myDate = new Date();
console.log(myDate);
console.log(myDate instanceof Date);
```

11.4. Mix HTML and JavaScript

```
<!DOCTYPE html>
<html>
<body>
  <h1>JavaScript in the HTML page</h1>
  <script>
    for (let i=1; i<=10; i++) {
      document.write(`<p>${i}</p>`);
```

```
    }
  </script>
</body>
</html>
```

На браузъра се изписва това:



JavaScript in the HTML page

1
2
3
4
5
6
7
8
9
10

11.5. Debugging Techniques

Strict Mode

- Strict mode limits certain "sloppy" language features
 - Silent errors will **throw Exception** instead – в началото на файла пишем този ред 'use strict'

Новите браузъри ще го приемат за команда, но старите ще го приемат просто за стринг

```
'use strict';          // File-Level
mistypeVariable = 17; // ReferenceError
```

```
function strict() {
  'use strict';        // Function-Level
  mistypeVariable = 17;
}
```

- Enabled by default in **modules**

11.6. Language Specifics

First-class Functions

- First-class functions – a function can be passed as an **argument** to other functions
- Can be **returned** by another function and can be **assigned** as a value to a variable

```
function running() {
  return "Running";
}
```

```

function category(fun, type) {
  console.log(fun() + " " + type); // с тези скоби () изпълняваме реално функцията running
}

category(running, "sprint"); //Running sprint

```

Nested Functions

- Functions can be **nested - hold other functions**
 - Inner functions have **access to variables** from **their parent**

```

function hypotenuse(m, n) { // outer function
  function square(num) { // inner function
    return num * num;
  }
  return Math.sqrt(square(m) + square(n));
}

```

Hoisting

- Variable and function declarations are **put into memory** during the **compile** phase, but stay exactly where you **typed** them in your code
- **Only declarations are hoisted**

Всички функции, декларирани декларативно се зареждат до нивото на своя scope при първото преминаване на **интерпретатора**. Т.е. вложена функция се hoist-ва до scope на функцията родител.

Всички var декларации също се hoist-ват.

Hoisting Variables

```

console.log(num); // Returns undefined
var num;
num = 6;
-----
num = 6;
console.log(num); // returns 6
var num;
-----
num = 6;
console.log(num); // ReferenceError: num is not defined
let num;
-----
console.log(num); // ReferenceError: num is not defined
num = 6;

```

Hoisting Functions

```

run(); // running
function run() {
  console.log("running");
};
-----

```

```

walk(); // ReferenceError: walk is not defined

let walk = function () {
  console.log("walking");
};

-----
console.log(walk); //undefined
walk(); // TypeError: walk is not a function
var walk = function () {
  console.log("walking");
};

```

12. Arrays and Nested Arrays

Можем да правим стек, опашка с масиви в JS.

Масивите в JS са като листовете в Java.

- Neither the **length** of a JavaScript array **nor** the **types** of its elements are **fixed**
- An array's **length can be changed** at any time
- Data can be stored at non-contiguous locations in the array
- JavaScript arrays are not guaranteed to be dense

12.2. Accessing Array Elements

Array elements are accessed using their **index**

```

let cars = ['BMW', 'Audi', 'Opel'];
let firstCar = cars[0];    //BMW
let lastCar = cars[cars.length - 1]; //Opel

```

`new Array(4).fill(properWheelSize, 0, 4);`//масив от 4ри елемента с value properWheelSize – попълни данните от нулев индекс до 4ти индекс без него

- Accessing indexes that do not exist in the array returns **undefined**

```

console.log(cars[3]);    // undefined
console.log(cars[-1]);  // undefined

```

Arrays can be **iterated** using **for-of** loop

```
for (let car of cars) { ... }
```

Arrays Indexation

Setting values via **non-integers** using **bracket notation** (or dot notation) creates **object properties** instead of array elements (will be discussed in later lesson) – създава свойство на масива/обект, а не елемент от масива

```

let arr = [];
arr[3.4] = 'Oranges'; // създава свойство на масива/обект, а не елемент от масива
arr[-1] = 'Apples'; // създава свойство на масива/обект, а не елемент от масива
console.log(arr.length);           // 0
console.log(arr.hasOwnProperty(3.4)); // true
arr["1"] = 'Grapes';

```

```
console.log(arr.length); // 2  
console.log(arr); // [ <1 empty item>, 'Grapes', '3.4': 'Oranges', '-1': 'Apples' ]
```

Destructuring Syntax

- Expression that **unpacks values from arrays or objects**, into distinct **variables**

```
let numbers = [10, 20, 30, 40, 50];  
let [a, b, ...elems] = numbers;  
Rest operator  
console.log(a) // 10  
console.log(b) // 20  
console.log(elems) // [30, 40, 50]
```

numbers и rest са различни референции/обекти – като правим операция по number, те не се отразяват в rest

- The **rest operator** can also be used to collect function parameters into an array

```
const [x, y] = "edno dve".split(" ");
```

Пример за деструктивен синтаксис:

Нормален синтаксис

```
'favourite_DownTown_3:14'  
let splitedArr = currentInput.split("_");  
let currentTypeList = splitedArr[0];  
let currentName = splitedArr[1];  
let currentTime = splitedArr[2];
```

Деструктивен синтаксис:

Като ги събираме на едно място е **rest** оператор.

Като ги разделяме е **spread** оператор.

```
function solve(...elements) {  
    console.log(elements); // връща масив [ 1, 2, 3, 4, 5 ]  
}  
solve(...[1, 2, 3, 4, 5]); // Прави го от колекция/масив на набор от изброени елементи.  
  
function solve(...elements) {  
    console.log(elements); // връща масив [ 1, 2, 3, 4, 5 ]  
}  
solve(1, 2, 3, 4, 5);
```

Един по-стар начин на запис

```
function solve() {  
    // const a = arguments();  
    console.log(arguments[0]); //връща първият елемент  
}
```

```

solve('cat', 42, function () {
    console.log('Hello world!');
});

const products = new Set();
Array.from(products); //прави го на масив
[...products].join(", "); /това също го прави на масив

```

Създаване копие на масив чрез destructing syntax

```

function firstLast(strings = []) {
    const first = Number([...strings].shift()); //създава копие на масив
    const last = Number([...strings].pop());

    return first + last;
}

console.log(firstLast(['20', '30', '40']));

```

12.3. Mutator Methods - Modify the Array

Pop

- Removes the **last element** from an array and returns that element
- This method **changes the length** of the array

```

let nums = [10, 20, 30, 40, 50, 60, 70];
console.log(nums.length); // 7
console.log(nums.pop()); // 70
console.log(nums.length); // 6
console.log(nums);      // [ 10, 20, 30, 40, 50, 60 ]

```

Push

- The **push()** method **adds one or more** elements to the **end** of an array and **returns** the new **length** of the array

```

let nums = [10, 20, 30, 40, 50, 60, 70];
console.log(nums.length); // 7
console.log(nums.push(80)); // 8 (nums.length)
console.log(nums); // [ 10, 20, 30, 40, 50, 60, 70, 80 ]

```

Shift

- The **shift()** method **removes** the **first element** from an array and **returns** that **removed element**
- This method **changes the length** of the array

```

let nums = [10, 20, 30, 40, 50, 60, 70];
console.log(nums.length); // 7

```

```
console.log(nums.shift()); // 10 (removed element)
console.log(nums); // [ 20, 30, 40, 50, 60, 70 ]
```

Unshift

- The **unshift()** method adds one or more elements to the beginning of an array and returns the new length of the array

```
let nums = [40, 50, 60];
console.log(nums.length); // 3
console.log(nums.unshift(30)); // 4 (nums.length)
console.log(nums.unshift(10,20)); // 6 (nums.length)
console.log(nums); // [ 10, 20, 30, 40, 50, 60 ]
```

Splice

- Changes the contents of an array by removing or replacing existing elements and/or adding new elements

```
let nums = [1, 3, 4, 5, 6];
nums.splice(1, 0, 2); // inserts at index 1 // изтрий нула на брой елемента считано от индекс 1, и добави/вмъкни елемент със стойност 2 след индекс 1.
```

```
console.log(nums); // [ 1, 2, 3, 4, 5, 6 ]
nums.splice(4,1,19); // replaces 1 element at index 4 // изтрий един на брой елемент считано от индекс 4, и добави/вмъкни елемент със стойност 19 след индекс 4.
```

```
console.log(nums); // [ 1, 2, 3, 4, 19, 6 ]
let el = nums.splice(2,1); // изтрий един на брой (само 1 път) елемент считано от индекс 2

console.log(nums); // [ 1, 2, 4, 19, 6 ]
console.log(el); // [ 3 ]
```

Fill

- Fills all the elements of an array from a start index to an end index with a static value

```
let arr = [1, 2, 3, 4];

// fill with 0 from position 2 inclusive until position 4 exclusive
console.log(arr.fill(0, 2, 4)); // [1, 2, 0, 0]

// fill with 5 from position 1 inclusive
console.log(arr.fill(5, 1)); // [1, 5, 5, 5]

// fill all elements of the array with 6
console.log(arr.fill(6)); // [6, 6, 6, 6]
```

Reverse

- Reverses the array
 - The first array element becomes the last, and the last array element becomes the first

```
let arr = [1, 2, 3, 4];
arr.reverse(); // не прави копие - директно го променя масива/обръща
```

```
console.log(arr); // [ 4, 3, 2, 1 ]
```

```
const oddNums = numbers
  .filter((value, index) => index % 2 == 1)
  .map(x => x * 2)
  .reverse();
```

Sorting Arrays

- The **sort()** method sorts the items of an array
- Depending on the provided **compare function**, sorting can be **alphabetic** or **numeric**, and either **ascending (up)** or **descending (down)**
- **By default**, the **sort()** function **sorts the values as strings** in **alphabetical** and **ascending order by ASCII codes**
- If you want to sort numbers or other values, you need to provide the correct **compare function!**
- **Винаги да слагаме функция в sort**

```
let numbers = [20, 40, 10, 30, 100, 5];
numbers.sort(); // Unexpected result on arrays of numbers!
console.log(numbers); // [10,100,20,30,40,5]
```

Compare Functions

A **function receiving two parameters**, e.g. **a** and **b**

- Returns either a **positive** number, a **negative** number, or **zero**
- If **result < 0**, **a** is sorted **before** **b** - descending
- If **result > 0**, **a** is sorted **after** **b** - ascending
- If **result = 0**, **a** and **b** are **equal** (no change)

Compare function for array of numbers

```
let nums = [20, 40, 10, 30, 100, 5];
nums.sort((a, b) => a-b); // Compare elements as numbers - Ascending
console.log(nums.join('|')); // 5|10|20|30|40|100
```

Compare function for array of strings

- The **localeCompare()** method is used to compare any two characters without regard for the case used
 - It's a string method so it can't be used directly on an array
 - Pass **localeCompare()** as the comparison function

```
let words = ['nest', 'Eggs', 'bite', 'Grip', 'jAw'];
words.sort((a, b) => a.localeCompare(b));
// ['bite', 'Eggs', 'Grip', 'jAw', 'nest']
```

12.4. Accessor Methods - methods return a new array

Join

- Creates and returns a **new string** by **concatenating** all of the elements in an array (or an array-like object), **separated** by commas or a **specified separator** string

```
let elements = ['Fire', 'Air', 'Water'];
console.log(elements.join()); // "Fire,Air,Water"
console.log(elements.join(' ')); // "FireAirWater"
console.log(elements.join('-')) // "Fire-Air-Water"
```

```
console.log(['Fire'].join(".")); // Fire
```

Concat

- The **concat()** method is used to **merge** two or more arrays
- This method **does not change** the **existing arrays**, but instead returns a new array

Копира стойностите в един нов масив

```
const num1 = [1, 2, 3];
const num2 = [4, 5, 6];
const num3 = [7, 8, 9];
const numbers = num1.concat(num2, num3);
console.log(numbers); // [1, 2, 3, 4, 5, 6, 7, 8, 9]
```

Slice

- The **slice()** method **returns** a shallow **copy** of a **portion** of an array into a **new array** object selected from begin to end (end not included)
- The **original array will not be modified**

```
let myArray = ["one", "two", "three", "four", "five"];
let sliced = myArray.slice(2); //индекс 2
console.log(myArray); //["one", "two", "three", "four", "five"]
console.log(sliced); // ["three", "four", "five"]
console.log(myArray.slice(2,4)); // ["three", "four"] //от индекс 2 вкл до индекс 4-1 = 3
```

Създаване копие на масив

```
let copyArray = array.slice(); - връща копие на масив/ на част от масива – копието е нова/ друга референция в паметта
let copyArray = array.slice(0); //същото
```

Includes

- Determines whether an array contains a certain element, returning **true** or **false** as appropriate

```
// array length is 3
// fromIndex is -100 - търси от index -100 до края на масива
// computed index is 3 + (-100) = -97
let arr = ['a', 'b', 'c'];
arr.includes('a', -100); // true
arr.includes('b', -100); // true
arr.includes('c', -100); // true
arr.includes('a', -2); // false
```

IndexOf

- The **indexOf()** method **returns** the **first index** at which a given **element** can be **found** in the array
 - Output is **-1** if element is **not present**

```
const beasts = ['ant', 'bison', 'camel', 'duck', 'bison'];
```

```
console.log(beasts.indexOf('bison')) // 1
// start from index 2
console.log(beasts.indexOf('bison', 2)) // 4
console.log(beasts.indexOf('giraffe')) // -1
```

12.5. Iteration Methods

forEach

- The **forEach()** method **executes a provided function** once for each array element
- Converting a for loop to forEach

```
const items = ['item1', 'item2', 'item3'];
const copy = [];
```

```
// For loop
for (let i = 0; i < items.length; i++) {
  copy.push(items[i]);
break; можем да кажем тук
}
```

```
// ForEach
items.forEach(item => { copy.push(item); }); //тук няма опция за break
```

```
// Arrow function
```

```
forEach((element) => { ... } )
forEach((element, index) => { ... } )
forEach((element, index, array) => { ... } )
```

.map

- **Creates a new array** with the results of calling a **provided function** on every element in the calling array

```
let numbers = [1, 4, 9];
let roots = numbers.map(function(num) {
  console.log(Math.sqrt(num));
  return Math.sqrt(num);
});
// roots is now [1, 2, 3]
// numbers is still [1, 4, 9] - след прилагането на .map първоначалния масив си остава същия
```

Some – ако има едно true, то целият израз ще върне true

- The **some()** method **tests** whether **at least one** element in the array passes the test implemented by the **provided function**
- It returns a **Boolean** value
- Функцията **some()** ще провери предиката, и ако има едно true, то целият израз ще върне **true**

```

let array = [1, 2, 3, 4, 5];
let isEven = function(element) {
  // checks whether an element is even
  return element % 2 === 0;
};

console.log(array.some(isEven)); //true

```

Find – и това май също е излишно

- Returns the **first found value** in the array, if an **element** in the array **satisfies** the **provided** testing function or **undefined** if not found

```

let array1 = [5, 12, 8, 130, 44];
let found = array1.find(function (element) {
  return element > 10;
});

console.log(found); // 12

```

Filter

- Creates a **new array** with **filtered elements only**
- Calls a **provided** callback **function** once for each element in an array
- **Does not mutate** the **array** on which it is called

Филтър с един критерий

```

let fruits = ['apple', 'banana', 'grapes', 'mango', 'orange'];
// Filter array items based on search criteria (query)
function filterItems(arr, query) {
  return arr.filter(function (el) {
    return el.toLowerCase().indexOf(query.toLowerCase()) !== -1;
  });
}

console.log(filterItems(fruits, 'ap')); // ['apple', 'grapes']

```

Филтър с два критерия

`predicate: (value: any, index: number, array: any[])`

12.6. Reducing Arrays

Reduce

- The **reduce()** method executes a reducer function on each element of the array, resulting in a **single output value**

```

const array1 = [1, 2, 3, 4];
const reducer =
(accumulator, currentValue) => accumulator + currentValue;

```

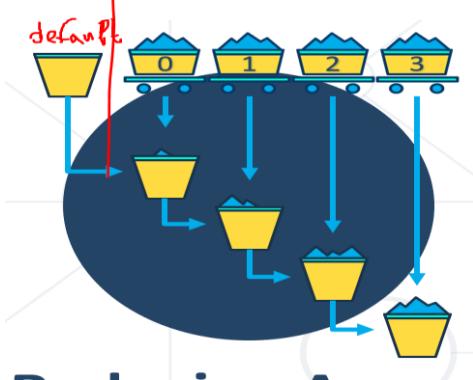
```
console.log(array1.reduce(reducer)); //10
console.log(array1.reduce(reducer, 5)); //15 //първоначално събира с 5
```

Reducer Function

- The reducer function takes **four** arguments:
 - Accumulator
 - Current Value
 - Current Index (Optional)
 - Source Array (Optional)
- Your **reducer function's** returned value is **assigned** to the **accumulator**
- Accumulator's value** - the **final, single** resulting **value**

Sum all values

```
let sum = [7, 1, 2, 3].reduce(function (acc, curr) {
  return acc + curr;
}, 0); - дефайлтна стойност е зададена като нула.
```



Ако не е зададена, то дефайлтната стойност за първи елемент е първия елемент на масива.

```
console.log(sum); // 13
```

Finding an average with reduce

```
const numbersArr = [30, 50, 40, 10, 70];
const average =
  numbersArr.reduce((total, number, index, array) => {
    total += number;
    if (index === array.length - 1) {
      return total / array.length;
    } else { return total; }
  });

console.log(average) // 40

//Вариант 2:
Array.prototype.average = function () {
  return this.reduce((acc, el) => {
    return acc + (el /this.length); //this тук сочи към прототипа на Array държавен
    конструктор функция / клас, т.е. за всяка инстанция на Array
    //сумираме дробната част на резултата „елемента делено на общия брой елементи“
}
```

```
    }, 0);  
}
```

Reduce and arrays

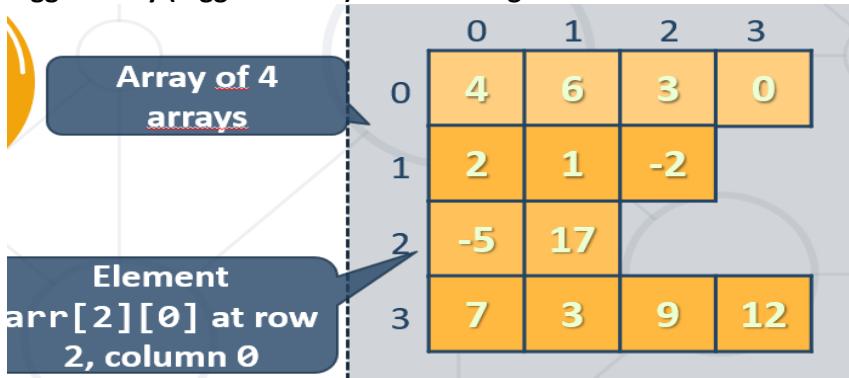
```
function incrSubs(arr = []) {  
  let biggest = Number.MIN_SAFE_INTEGER;  
  let result = [];  
  // const result = arr.filter(el => {  
  //   if (el >= biggest) {  
  //     biggest = el;  
  //     return true;  
  //   }  
  //   return false;  
  // });  
  
  arr.reduce((accumulated, current) => {  
    if (current >= biggest) {  
      biggest = current;  
      accumulated.push(current);  
    }  
  
    return accumulated;  
  }, result); //we inject here an empty array which at the end is the accumulated.  
  
  return result;  
}
```

Reduce and objects

```
const formData = new FormData(event.target); //масив от DOM елементи  
//на всеки ключ/стойност от масива, го сложи в обект с ключ стринг стойността на [k] и value  
v = от масив на асоциативен масив. Ако не го деструктурираме, ще получим пропърти с буквa k.  
//entries() връща итератор/stream, и го правим наново на масив, и чак след това на обект  
const data = [...formData.entries()].reduce((a, [k, v]) => Object.assign(a, {[k]: v}), {});  
//извади стринга, който се крие в k
```

12.7. Array of Arrays – nested arrays

Jagged array (Jagged matrix) – not rectangular



The diagram shows a 4x4 grid where each row has a different number of columns. Row 0 has 4 columns (4, 6, 3, 0). Row 1 has 3 columns (2, 1, -2). Row 2 has 2 columns (-5, 17). Row 3 has 4 columns (7, 3, 9, 12). A callout bubble points to the first row with the text "Array of 4 arrays". Another callout bubble points to the element at row 2, column 0 with the text "Element arr[2][0] at row 2, column 0".

	0	1	2	3
0	4	6	3	0
1	2	1	-2	
2	-5	17		
3	7	3	9	12

```
let arr = [  
  [4, 6, 3, 0],  
  [2, 1, -2],  
  [-5, 17],  
  [7, 3, 9, 12]  
];
```

```
for (let row of arr) {
    for (let col of row) {
        console.log(col);
    }
}

for (let row of arr) {
    console.log(row.join(" "));
}

let arr = [[4, 5, 6],
           [6, 5, 4],
           [5, 5, 5]];

arr.forEach(printRow);

function printRow(row) {
    // console.log(row);
    row.forEach(printNumber);
}

function printNumber(num) {
    console.log(num);
};

function solve(matrix) {
    let sumMainDiagonal = 0;
    let sumSecondDiagonal = 0;

    for (let row = 0; row < matrix.length; row++) {
        sumMainDiagonal += matrix[row][row];
    }

    for (let row = matrix.length - 1, col = 0; row >= 0; row--, col++) {
        sumSecondDiagonal += matrix[row][col];
    }

    console.log(sumMainDiagonal + " " + sumSecondDiagonal);
}

.flat()
const arr1 = [0, 1, 2, [3, 4]];

console.log(arr1.flat());
```

```
// expected output: [0, 1, 2, 3, 4]

const arr2 = [0, 1, 2, [[[3, 4]]]];
console.log(arr2.flat(2));
// expected output: [0, 1, 2, [3, 4]]
```

13. Objects & Composition

13.1. Objects in JavaScript

What is an Object?

- An object is a **collection of fields**, called **properties**
- A property is an association between a name (or **key or identifier**) and a **value**
- Objects are a **reference data type**

In JavaScript they are created with an **object literal**:

```
const person = {
  firstName: "John",
  lastName: "Doe",
  age: 50
};
```

Object Properties

- A **property** of an object can be explained as a **variable** that is **attached** to the object
- Object properties are the same as **ordinary variables**, and can hold **any data type** and be **reassigned**

Assigning and Accessing Properties

Simple dot-notation – constant name of the property

```
const person = { name: 'Peter' };
console.log(person.name); // Peter
```

Bracket-notation (indexing operator)

- Required if the key contains a **special character**

```
let person = {};
person['job-title'] = 'Trainer';
console.log(person['job-title']); // Trainer
console.log(person.job-title) // ReferenceError
```

Option for creating property names/keys dynamically

```
const b = 'B';
const c = 'C';

const data = {
  a: true,
  [b]: true, // dynamic property
  [`interpolated-${c}`]: true, // dynamic property + interpolation
  [`${b}-${c}`]: true
```

```
}
```

```
console.log(data);
```

- Brackets can be used with keys as **string variables**

Вариант 1 - работи:

```
let person = {
    name : 'Ivan',
    age : 36,
}
console.log(person.name); //Ivan
console.log(person['name']); //Ivan
```

Вариант 2 - работи:

```
let person = {
    'name' : 'Ivan',
    age : 36,
}
console.log(person.name); //Ivan
console.log(person['name']); //Ivan
```

- Properties can be **added** during run-time

```
const person = { name: 'Peter' };
person.age = 21; // { name: 'Peter', age: 21 }
console.log(person.age); // 21
```

- Unassigned properties of an object are **undefined**

```
const person = { name: 'Peter' };
console.log(person.lastName); // undefined
```

Съкратен запис на JavaScript

Когато ключовете съвпадат по имена с параметрите на функцията, можем да запишем съкратено:

```
function solve(name, population, treasury) {
    return result = {
        name,
        population,
        treasury
    }
}

function solve(name, population, terasury) {
    return result = {
        name: name,
        population: population,
        treasury: terasury,
```

```

    }
}

console.log(
  solve('Tortuga',
    7000,
    15000
));

```

Destructuring Syntax in Objects

- "Dive into" an **object** and extract properties by name
- Can be used to get **multiple** property values
- Не спазваме реда като при масивите, но спазваме имената на свойствата

```

const department = {
  name: 'Engineering',
  director: 'Ted Thompson',
  employeeCount: 25
};
const { name, employeeCount } = department;
console.log(name, employeeCount); // 'Engineering' 25

```

Deleting Properties

```

const person = {
  name: 'Peter',
  age: 21,
  ['job-title']: 'Trainer'
}
// Object {name: 'Peter', age: 21, 'job-title': 'Trainer }

delete person.age;
// Object {name: 'Peter', 'job-title': 'Trainer }

console.log(person.age) // undefined

```

Object References

- Variables holding **reference** data types contain the **memory address** (reference) of the data
- **Copies** of the reference point to the **same data**



```

let x = {name: 'John'};
let y = x;

```

```
y.name = 'Steve';
console.log(x.name); // Steve
```

Comparing Objects

- Two variables, **two distinct objects** with the same properties

```
const fruit = {name: 'apple'};
const fruitbear = {name: 'apple'};
fruit == fruitbear; // false
fruit === fruitbear; // false
```

- Two variables, a **single object**

```
const fruit = { name: 'apple' };
const fruitbear = fruit;
// Assign a copy of the fruit reference to fruitbear
fruit == fruitbear; // true
fruit === fruitbear; // true
```

13.2. Objects as Associative Arrays

13.2.1. Associative Arrays

- Objects can serve the role of **associative arrays** in JavaScript
 - The keys (property names) are **string indexes**
 - Values are **associated** to a key
 - All values should be of the **same type**

For... in Loop

- **for ... in** – iterates over all **enumerable** properties

```
const obj = {a: 1, b: 2, c: 3};
for (const key in obj) {
  console.log(`obj.${key} = ${obj[key]}`);
}
// Output:
// "obj.a = 1"
// "obj.b = 2"
// "obj.c = 3"
```

Object Keys and Values and Entries

- Obtain an **array** of all **keys** or **values** in an object:

```
const phonebook = {
  'Tim': '555-111',
  'Bill': '555-333',
  'Peter': '555-777'
};
const keys = Object.keys(phonebook);
console.log(keys);
// ['Tim', 'Bill', 'Peter']

const values = Object.values(phonebook);
```

```
console.log(values);
// ['555-111', '555-333', '555-777']
```

- Get an array of **tuples** (array of two elements), representing each key and value pair
 - First tuple element is the **key**, the second is the **value** – прави го на масив, като всеки елемент е масив от ключ и стойност

```
const entries = Object.entries(phonebook);
console.log(entries);
// [ ['Tim', '555-111'],
//   ['Bill', '555-333'],
//   ['Peter', '555-777'] ]
```

- This method is often used if we want to **sort** the contents

```
const towns = {};
for (let townAsString of townsArr) {
    let [name, population] = townAsString.split(' <-> ');
    population = Number(population);
    if (towns[name] != undefined) {
        population += towns[name];
    }
    towns[name] = population;
}
```

13.3. Methods and Context

Object Methods

- Objects can also have **methods**
- Methods are **actions** that can be performed on objects
- Methods are stored in **properties** as **function** definitions

```
let person = {
  firstName: "John",
  lastName: "Doe",
  age: function (myAge) {
    return `My age is ${myAge}!`
  }
  age(myAge) { //съкратен запис
    return `My age is ${myAge}!`
  }
};

console.log(person.age(21)); // My age is 21!
```

Objects as Function Libraries ~ като Статичен клас в JAVA

- Related functions may be **grouped** in an object
- The object serves as a **function library**
 - Similar to built-in libraries like **Math**, **Object**, **Number**, etc.

```
// sorting helper
const compareNumbers = {
  ascending: (a, b) => a - b;
  descending: (a, b) => b - a;
};
```

- This technique is often used to **expose public API** in a module

Objects as switch replacement

- You will **almost never** see **switch** used in JS code
- **Named methods** are used instead

В практиката се ползва по-малко switch, защото ако изтървем един break, и положението отиде...

```
let count = 5;
switch (command) {
  case 'increment':
    count++;
    break;
  case 'decrement':
    count--;
    break;
  case 'reset':
    count = 0;
    break;
}
```

Shorter syntax for object methods

```
let count = 5;
const parser = {
  increment() { count++; },
  decrement() { count--; },
  reset() { count = 0; }
}
parser[command]();
```

Accessing Object Context

- Functions in JavaScript have **execution context**
 - Accessed with the keyword **this**
 - When executed as an **object method**, the context is a reference to the **parent object**

```
const person = {
  firstName: 'Peter',
  lastName: 'Johnson',
  fullName() {
    return this.firstName + ' ' + this.lastName;
  }
};
console.log(person.fullName()); // 'Peter Johnson'
```

Function Execution Context

- Execution context can be **changed** at run-time
- If a function is **executed outside** of its parent object, it will **no longer** have access to the object's content

Функция на един обект, извън обсега на този обект може да се извика на съвсем друг обект, но резултата от нея ще бъде с данните на другия обект!

```
const person = {
  firstName: 'Peter',
```

```

lastName: 'Johnson',
fullName() {
    return this.firstName + ' ' + this.lastName;
}
};

const getFullName = person.fullName();
console.log(getFullName); // Peter Johnson

const anotherPerson = {
    firstName: 'Bob',
    lastName: 'Smith'
};

anotherPerson.fullName = person.fullName;
console.log(anotherPerson.fullName()); // 'Bob Smith'

console.log(person.fullName == anotherPerson.fullName); //true

```

- Further lessons will explore more context features!

13.4. Object Composition

Пример за обект с фиксиран параметър/променлива, и методи в самия обект

```

function cityTaxes(name, population, treasury) {
    let result = {
        name,
        population,
        treasury,
        taxRate : 10,
        collectTaxes() {
            this.treasury += Math.floor(this.population * this.taxRate);
        },
        applyGrowth(percentage) {
            this.population += Math.floor (this.population * percentage / 100);

        },
        applyRecession(percentage) {
            this.treasury -= Math.ceil(this.treasury * percentage / 100);
        }
    }
    return result;
}

```

Пример за вложени обекти:

```
function solve(arr = []) {
```

```

let catalogue = {};
arr.forEach(el => {
    let [town, product, price] = el.split(" | ");
    price = Number(price);
    if (!catalogue.hasOwnProperty(product)) {
        catalogue[product] = {};
    }
    catalogue[product][town] = price;
})
return catalogue;
}

console.log(
solve(['Sample Town | Sample Product | 1000',
    'Sample Town | Orange | 2',
    'Sample Town | Peach | 1',
    'Sofia | Orange | 3',
    'Sofia | Peach | 2',
    'New York | Sample Product | 1000.1',
    'New York | Burger | 10']
));

```

Combining simple objects into more complex ones

Пример за обект, в който има друг обект

```

let student = {
    firstName: 'Maria',
    lastName: 'Lopez',
    age: 22,
    location: { lat: 42.698, lng: 23.322 }
}
console.log(student);
console.log(student.location.lat);

```

- **Composition** is a powerful technique for **code reuse**
- It can be considered **superior** to **OOP inheritance**

Composing Objects

```

let name = "Sofia";
let population = 1325744;
let country = "Bulgaria";
let town = { name, population, country };
console.log(town); // Object {name: "Sofia", population: 1325744,
country: "Bulgaria"}

town.location = { lat: 42.698, lng: 23.322 };

```

```
console.log(town); // Object {..., location: Object}
```

Nested Destructuring

- Destructuring can work **beyond the top level** – работи на всички нива на обекта

```
const department = {
  name: "Engineering",
  dataObject: {
    director: {
      name: 'John',
      position: 'Engineering Director'
    },
    employees: [],
    company: 'Quick Build'
  }
};

const { dataObject: { director } } = department;
console.log(director); // director is { name: 'John', position: 'Engineering Director'}
```

Object and Array Destructuring

Масив от обекти

```
const employees = [
  { name: 'John', position: 'worker' }, // first element
  { name: 'Jane', position: 'secretary' }
];
const [{ name }] = employees;
console.log(name); // name of first element is 'John'
```

Обект от масиви

```
const company = {
  employees: ['John', 'Jane', 'Sam', 'Suzanne'],
  name: 'Quick Build',
};
const { employees: [empName] } = company;
console.log(empName); // empName is 'John'
```

Връщаме като обект от методи за изпълнение

```
function calculator() {
  let firstInput;
  let secondInput;
  let resultBox;

  function init(selector1, selector2, selector3) {
    firstInput = document.querySelector(selector1);
    secondInput = document.querySelector(selector2);
    resultBox = document.querySelector(selector3);
```

```

        console.log(firstInput, secondInput, resultBox);
    }

    function add() {
        resultBox.value = Number(firstInput.value) + Number(secondInput.value);
    }

    function subtract() {
        resultBox.value = Number(firstInput.value) - Number(secondInput.value);
    }

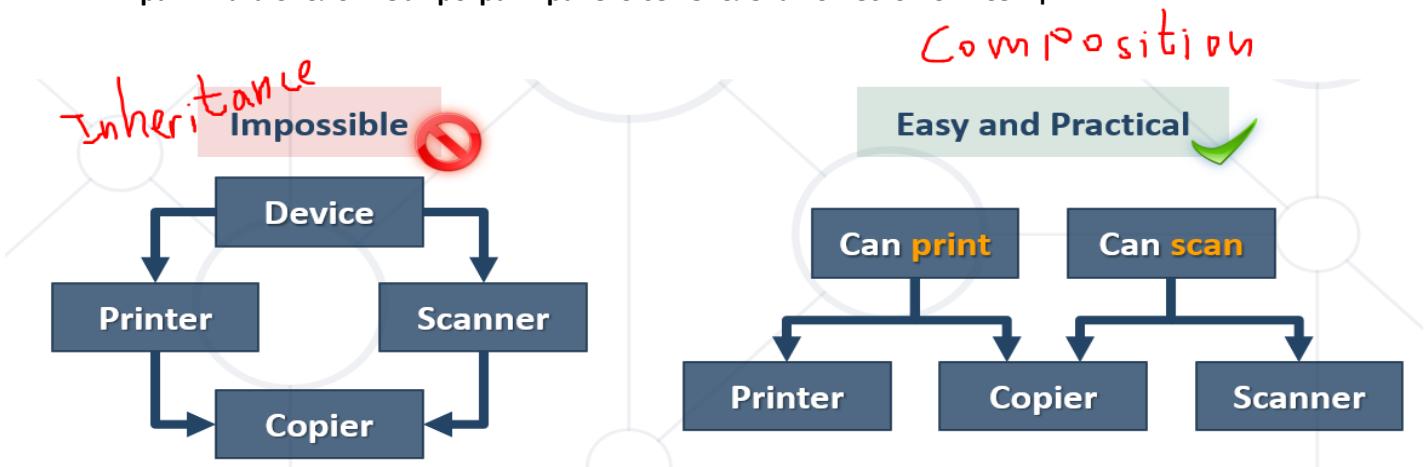
    return { init, add, subtract }; //връщаме като обект от методи за изпълнение
}

const calculate = calculator();
calculate.init('#num1', '#num2', '#result'); //извикваме метода init
calculate.add()
calculate.subtract()

```

Composing Objects with Behavior

- We can **compose behavior** at run-time and **reuse functionality**
- **Describe objects** in terms of what they **do**, not what they **are**
- This solves a deeply rooted **problem** with **OOP inheritance**
- В практиката около Web програмирането се използва по-често Композиция



Пример за композиция

```

function print() {
    console.log(` ${this.name} is printing a page`);
}

function scan() {
    console.log(` ${this.name} is scanning a document`);
}

const printer = {
    name: 'ACME Printer',
    print
};

```

```

printer.print(); // ACME Printer is printing a page

const scanner = {
  name: 'Initech Scanner',
  scan
};
scanner.scan(); // Initech Scanner is scanning a document

const copier = {
  name: 'ComTron Copier',
  print,
  scan
};
copier.print(); // ComTron Copier is printing a page
copier.scan(); // ComTron Copier is scanning a document

```

Factory Functions

- Functions that **compose** and **return** objects

```

function createRect(width, height) {
  const rect = { width, height };

  rect.getArea = () => {
    return rect.width * rect.height;
  };

  return rect;
}

```

- Creating objects with factory functions can **avoid** the pitfalls of using **this**

Интересна примерна задача:

```

function factory(library = {}, orders = []) {
  return orders.map((order) => compose(order));

  function compose(order) {
    //Create empty object
    //Copy properties from template
    const result = Object.assign({}, order.template); //задай на празен обект на
поръчката template-а.

    //Compose methods from library for every item in parts
    for (const part of order.parts) {
      result[part] = library[part];
    }

    //return result
    return result;
  }
}

```

```

}

const library = {
  print: function () {
    console.log(` ${this.name} is printing a page`);
  },
  scan: function () {
    console.log(` ${this.name} is scanning a document`);
  },
  play: function (artist, track) {
    console.log(` ${this.name} is playing '${track}' by ${artist}`);
  },
};

const orders = [
  {
    template: { name: 'ACME Printer' },
    parts: ['print']
  },
  {
    template: { name: 'Initech Scanner' },
    parts: ['scan']
  },
  {
    template: { name: 'ComTron Copier' },
    parts: ['scan', 'print']
  },
  {
    template: { name: 'BoomBox Stereo' },
    parts: ['play']
  },
];

```

const products = factory(library, orders);
 console.log(products);

Пример:

```

Object.assign(storage, remainingStorage);
1/4
assign(target: { carbohydrate: number; flavour: number; fat: number; protein: number; }, source: {}): { carbohydrate: number; flavour: number; fat: number; protein: number; }

```

Decorator Functions

- Functions that **add new data and behavior** to objects

```

function canPrint(device) {
  device.print = () => {
    console.log(` ${device.name} is printing a page`); //тук не използваме this
}

```

```

    }
}

const printer = { name: 'ACME Printer' };
canPrint(printer);
printer.print(); // ACME Printer is printing a page

```

- The object reference is **embedded**(украсено) – using **this** is not required

13.5. JSON - JavaScript Object Notation

- It's a **data interchange format**
- It's **language independent** - syntax is like JavaScript object syntax, but the JSON format is text only
- Is "**self-describing**" and easy to understand:

```
{
  "employees": [ //свойство задължително с двойни кавички
    { "firstName": "John", "lastName": "Doe" },
    { "firstName": "Anna", "lastName": "Smith" },
    { "firstName": "Peter", "lastName": "Jones" }
  ]
}
```

Syntax Rules

- In JSON:
 - Data is in **name/value pairs**
 - Data is **separated by commas**
 - **Curly braces** hold **objects**
 - **Square brackets** hold **arrays**
 - JSON only takes **double quotes** ""

```
{
  "employees": [{ "firstName": "John", "lastName": "Doe" }]
}
```

Parsing from Strings - to convert the JSON format into a JavaScript object

- A common use of JSON is to read data from a web server, and display the data in a web page
- Use the JavaScript built-in function `JSON.parse()` to convert the JSON format into a JavaScript object:

Функциите не се сериализират – не се пренасят по мрежата

```
let data = '{ "manager": {"firstName": "John", "lastName": "Doe"} }';
let obj = JSON.parse(data);
console.log(obj.manager.lastName) // Doe
```

Converting to String – from Object or Arrays format into String

- Use `JSON.stringify()` to convert objects into a string:

```
let obj = { name: "John", age: 30, city: "New York" };
let myJSON = JSON.stringify(obj);
console.log(myJSON); // {"name": "John", "age": 30, "city": "New York"}
```

- You can do the same for arrays

```
let arr = [ "John", "Peter", "Sally", "Jane" ];
let myJSON = JSON.stringify(arr);
console.log(myJSON); // ["John","Peter","Sally","Jane"]
```

- Format the string with **indentation** for readability

```
let arr = [ "John", "Peter", "Sally", "Jane" ];
let myJSON = JSON.stringify(arr, null, 2);
console.log(myJSON);
[
  "John",
  "Peter",
  "Sally",
  "Jane"
]
```

13.6. Cloning objects

There is a problem in JS with making the so called deep copy. In most cases, we can not make easily a deep copy.

Shallow copy with spread

A **shallow copy** of an object is a copy whose properties share the same references (point to the same underlying values) as those of the source object from which the copy was made. As a result, when you change either the source or the copy, you may also cause the other object to change too — and so, you may end up unintentionally causing changes to the source or copy that you don't expect. That behavior contrasts with the behavior of a [deep copy](#), in which the source and copy are completely independent.

```
let objExample = {
  a: 1,
  b: 2,
  c: {
    name: 'Simo',
    company: 'Blubito'
  }
};

// Shallow copy with spread
const objClone = { ...objExample };
objClone.d = true;
objClone.c.experience = 5;
console.log(objExample);
{ a: 1, b: 2, c: { name: 'Simo', company: 'Blubito', experience: 5 } }

console.log(objClone);
{ a: 1, b: 2, c: { name: 'Simo', company: 'Blubito', experience: 5 }, d: true }
```

Уж клонирано, ама пропърти "с" като му изменим съдържанието, и на двете места се записва

Clone with Object.assign

Работи много като клонирането със spread оператора!!! Т.е. не е дълбоко копие

```

const target = { a: 1, b: 2 };
const source = { b: 4, c: 5 };

const returnedTarget = Object.assign(target, source);

console.log(target);
// expected output: Object { a: 1, b: 4, c: 5 }

console.log(returnedTarget);
// expected output: Object { a: 1, b: 4, c: 5 }

```

Deep Copy Clone with JSON – no option for functions

```

let objExample = {
  a: 1,
  b: 2,
  c: {
    name: 'Simo',
    company: 'Blubito'
  }
};

```

```

// Clone with JSON – не работи обаче с пренасяне на функции
const objClone = JSON.parse(JSON.stringify(objExample));
objClone.d = true;
objClone.c.experience = 5;
console.log(objExample);
{ a: 1, b: 2, c: { name: 'Simo', company: 'Blubito' } }

console.log(objClone);
{ a: 1, b: 2, c: { name: 'Simo', company: 'Blubito', experience: 5 }, d: true }

```

13.7. Objects and reduce()

```

const furniture = Array.from(document.querySelectorAll('input[type="checkbox"]:checked'))
  .map(x => x.parentElement.parentElement)
  .map(r => ({ // слагаме обикновени скоби за да върнем поток от обекти, иначе с къдрави
    те скоби само ще иска return
    name: r.children[1].textContent,
    price: Number(r.children[2].textContent),
    decFactor: Number(r.children[3].textContent),
  }))
  .reduce((accumulator, currentValue, currIndex, initialArray) => {
    accumulator.names.push(currentValue.name);
    accumulator.total += currentValue.price;
  })

```

```

    accumulator.decFactor += currentValue.decFactor / initialArray.length;

    return accumulator;
}, { names: [], total: 0, decFactor: 0 }); //initial value

```

14. DOM (Document Object Model) Introduction

14.1. Browser API (Application Programming Interface)

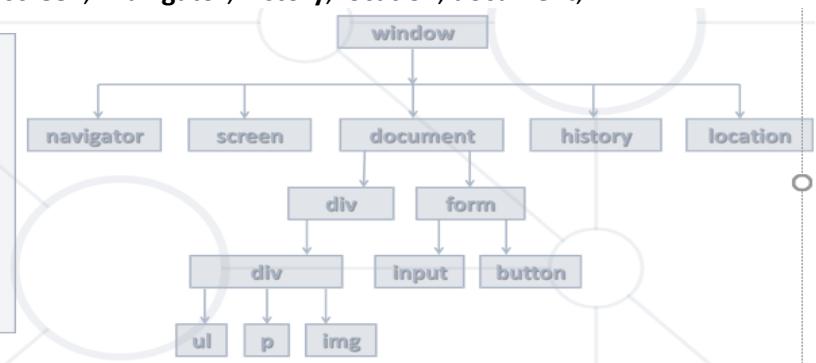
Browser Object Model (BOM)

- Browsers expose some objects like **window**, **screen**, **navigator**, **history**, **location**, **document**, ...

```

console.dir(window);
console.dir(navigator);
console.dir(screen);
console.dir(location);
console.dir(history);
console.dir(document);

```



- Most of this API will be examined in the **next course**

Global Context in the Browser

- The **global object** in the browser is **window**

```

> let b = 8;
  console.log(this.b);
undefined

```

```

function foo() {
  console.log("Simple function call");
  console.log(this === window); // true - window е глобалната променлива
}
foo();

```

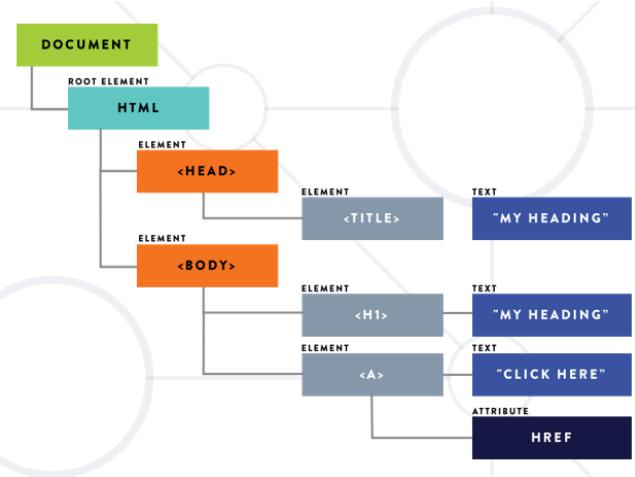
14.2. Document Object Model (DOM)

- The **DOM** represents the document as **nodes** and **objects**
 - That way, the programming languages **can connect** to the page
- The **HTML DOM** is an **Object Model** for **HTML**. It defines:
 - HTML elements as **objects**
 - Properties**
 - Methods**
 - Events**

From HTML to DOM Tree

- The browser **parses** HTML and creates a **DOM Tree**

```
<html>
  <head>
    <title>My Heading</title>
  </head>
  <body>
    <h1>My Heading</h1>
    <a href="/about">Click Here</a>
  </body>
</html>
```



- The elements are **nested** in each other and create a **hierarchy**
 - Like the hierarchy of a **street address** – Country, City, Street, etc.

- HTML DOM **method** is an action you can do (like **add** or **delete** an HTML element)

```
<!doctype html>
...<html> == $0
  <head>
    <title>Intro to DOM</title>
  </head>
  <body>
    <h1>Introduction to DOM</h1>
    <ul>
      <li>DOM Methods example</li>
      <li>DOM Properties example</li>
    </ul>
  </body>
</html>
```

```
>
let h1Element = document.getElementsByTagName('h1')[0];
console.log(h1Element);
<h1>Introduction to DOM</h1>
```

- HTML DOM **property** is a value that you can **get** or **set** (changing the content of an HTML element)

```
<!doctype html>
...<html> == $0
  <head>
    <title>Intro to DOM</title>
  </head>
  <body>
    <h1>Introduction to DOM</h1>
    <ul>
      <li>DOM Methods example</li>
      <li>DOM Properties example</li>
    </ul>
  </body>
</html>
```

```
let secondLi = document.getElementsByTagName('li')[1];
secondLi.innerHTML += " - DONE"
```

Introduction to DOM

- DOM Methods example
- DOM Properties example - DONE

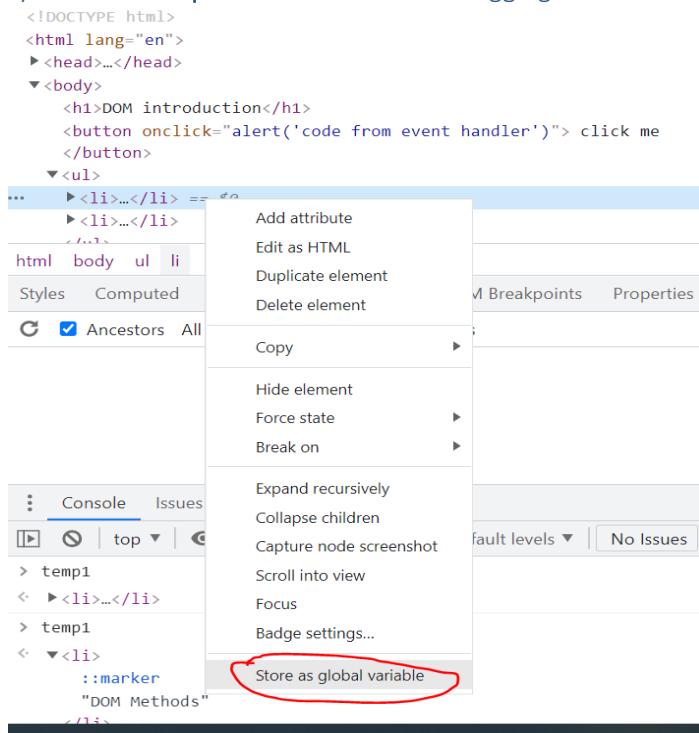
DOM Methods

- **DOM Methods - actions** you can perform on HTML elements
- **DOM Properties** - values of HTML elements that you can **set** or **change**
- JavaScript can **interact** with web pages via the **DOM API**:
 - Check the **contents** and **structure** of elements on the page
 - Modify element **style** and **properties**
 - Read **user input** and react to **events**
 - Create and remove elements
- Most actions are performed when an **event** occurs
 - Events are "**fired**" when something of interest happens
- All of this **and more** will be examined in upcoming lessons

JavaScript in the Browser

Code can be **executed** in the page in different ways:

1. Directly in the developer console – when debugging



2. As a page event handler – e.g., user clicks on a button

Вариант на **onclick** събитие върху бутон или друг елемент
console.log('Hello, DOM!') е реално JavaScript код.

onclick в HTML

```
<button onclick="console.log('Hello, DOM!')>Click Me</button> event
```

Вариант на **submit** събитие върху формулар form

```
<form name="myForm" action="/action_page.php" onsubmit="return validateForm()" method="post">
  Name: <input type="text" name="fname">
  <input type="submit" value="Submit">
```

```
</form>
```

Name: _____

Submit

3. Via inline script, using `<script>` tags

```
<script>
  function sum(a, b) {
    let result = a + b;
    return result;
  }
</script>
```

4. By importing from external file – most flexible method

```
<script src="demo.js"> </script>
```

JS demo.js X < index.html

JS demo.js > ⌂ add

```
1  function add(a, b) {
2    return a+b;
3 }
```

```
<script src="app.js"></script> //HTML

window.addEventListener('load', solve);

function solve() {
  window.addEventListener('load', solve); //JavaScript
  function solve() {
....}
}
```

14.3. HTML Elements

DOM Properties and HTML Attributes

Elements and Properties

- The DOM Tree is comprised of **HTML elements**
- Elements are **JS objects** with **properties** and **methods**
 - They can be **accessed** and **modified** like regular objects
- To change the contents of the page:
 - **Select** an element to obtain a **reference**
 - **Modify** its **properties**

Attributes and Properties

- Attributes are defined by **HTML**
 - Attributes **initialize** DOM properties
 - **Property** values can **change** via the DOM API
- The **HTML attribute** and the **DOM property** are technically **not the same thing**
- Since the **outcome is the same**, in practice you will **almost never** encounter a difference!

Дали ще променим атрибут или свойство, резултатът е едно и също!

Чрез използване на DOM свойствата не можем да изтрием DOM свойство от HTML елемент.

Чрез използване на HTML атрибути можем да изтрием HTML атрибут.

Пример:

При използване на HTML атрибути

```
<input type="text" class="my-class" name="username" placeholder="Username..."/>
<input type="password" class="my-class" name="password" placeholder="Password..."/>
```

```
const inputPassEle = document.getElementsByTagName('input')[1];
inputPassEle.removeAttribute('placeholder');
inputPassEle.removeAttribute('class'); //тук изтриваме класа като атрибут
```

```
<input type="text" class="my-class" name="username" placeholder="Username..."/>
<input type="password" name="password"/>
```

При използване на DOM свойства:

```
<input type="text" class="my-class" name="username" placeholder="Username..."/>
<input type="password" class="my-class" name="password" placeholder="Password..."/>
const inputPassEle = document.getElementsByTagName('input')[1].classList;
console.log(inputPassEle);
inputPassEle.remove('my-class');
```

```
<input type="text" class="my-class" name="username" placeholder="Username..."/>
<input type="password" class name="password" placeholder="Password..."/> //не е изтрито
```

DOM Manipulations

Accessing a DOM element is an expensive operation for the processor!!!

- The **HTML DOM** allows JavaScript to change the content of **HTML elements**
 - **innerHTML**
 - **textContent**
 - **value**
 - **style**
 - And many others to be discussed in upcoming lessons

Accessing Element HTML

Целият content на даден елемент, или с други думи от `<div>` включително до `</div>` включително. Така е опасно да работим тъй като браузърът го чете като HTML код, и някоя потребител може да въведе `<p>I am hacker</p>`, което да се изпише на страницата 😊

- To access raw HTML:

```
element.innerHTML = "<p>Welcome to the DOM</p>";
```

```
<html>
  <head></head>
  <body>
    <div id="main">This is JavaScript!</div>
  </body>
</html>
```

```
<html>
  <head></head>
  <body>
    <div id="main">
      <p>Welcome to the DOM</p>
    </div>
  </body>
</html>
```

- This will be parsed – beware of **XSS attacks!** – Най-високо ниво на достъп браузърът дава на кода от сървъра, а най-ниско на код от конзола
- Changing `textContent` or `innerHTML` removes all child nodes – изтрива всички деца/поделементи

Accessing Element Text – за статични полета

- The contents of HTML elements are stored in text nodes
 - To access the contents of an element:

.content взима escape-нат текст – Ако някой потребител въведе `<p>I am hacker</p>`, то няма да се изпише на страницата

```
let text = element.textContent; //This is JavaScript!
element.textContent = "Welcome to the DOM";
```

```
<html>
  <head></head>
  <body>
    <div id="main">This is JavaScript!</div>
  </body>
</html>
```

```
<html>
  <head></head>
  <body>
    <div id="main">Welcome to the DOM</div>
  </body>
</html>
```

- If the element has children, returns all text concatenated

Accessing Element Values – за динамични полета – къдемо потребителят въвежда данни, като `<input type="text">` например или като `<textarea id="input" cols="30" rows="12"></textarea>`

- The values of input elements are string properties on them:

```

<html>
  <head></head>
  <body>
    <div id="main">
      <p>Welcome to the DOM</p>
      <input id="num1" type="text">
    </div>
  </body>
</html>

```

```

type: "text"
useMap: ""
validationMessage: ""
▶ validity: ValidityState
value: "56"
valueAsNumber: NaN
▶ webkitEntries: Array[0]
webkitdirectory: false
width: 0

```

```

let num = Number(element.value);
element.value = 56;

```

Задача:

```

<script>
  /**
   * @param {string} element
   */
  function edit(element, match, replacer) {
    const text = element.textContent;
    element.textContent = text.split(match).join(replacer);
  }
</script>

```

14.4. Targeting DOM Elements

Obtaining Element References

Targeting Elements

- There are a few ways to **find** a certain **HTML element** in the **DOM**:
 - By ID - **getElementById()**
 - By class name - **getElementsByClassName()**
 - By tag name - **getElementsByTagName()**
 - By CSS selector - **querySelector()**, **querySelectorAll()**
- These methods return a **reference** to the element, which can be **manipulated** with JavaScript

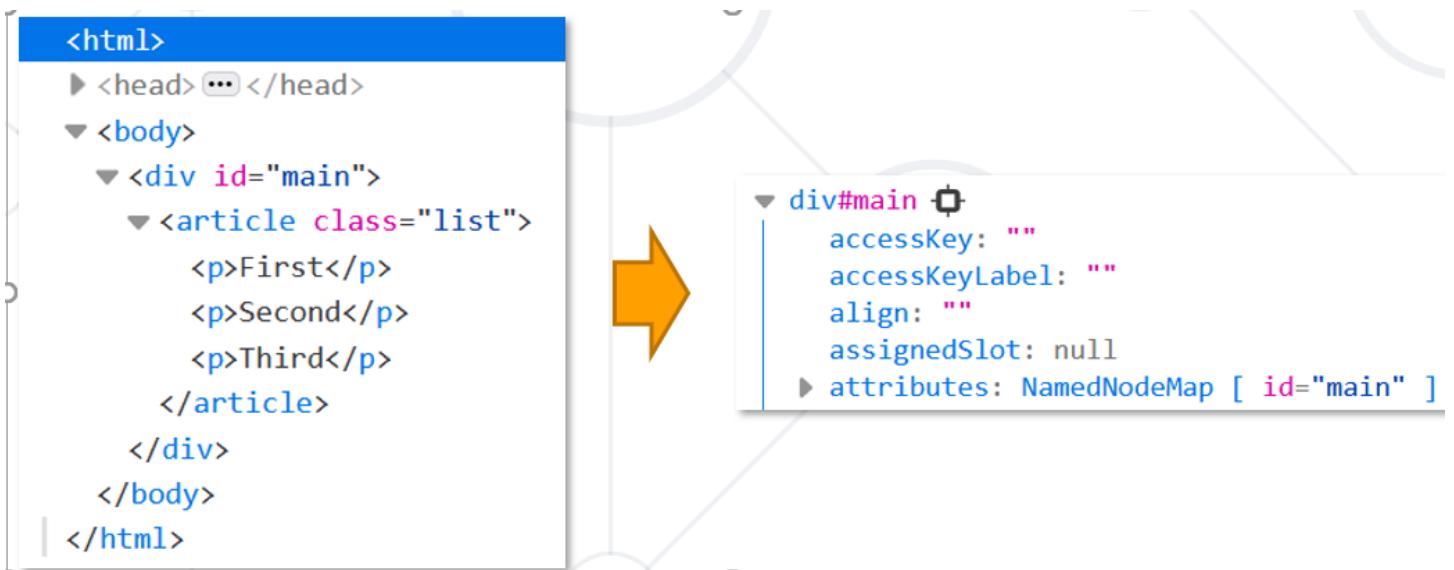
Targeting by ID – Example – връща HTMLCollection

- The **ID attribute** must be **unique** on the page – IDто е референция към елемента

```

const element = document.getElementById('main');
console.log(element);

```



Targeting by Tag and Class Names - връща HTMLCollection

- The **tag name** specifies the **type** of element – **div**, **p**, **ul**, etc.

```
const elements = document.getElementsByTagName('p');
//document e Select all paragraphs on the page
```

- Class names** are used for **styling** and easier **selection**

```
const elements = document.getElementsByClassName('list');
// Select all elements having a class named 'list'
```

- Both methods return a **live HTMLCollection**
 - Even if only **one** element is selected! This is a **common mistake**

CSS Selectors – връща NodeList

- CSS selectors** are strings that follow CSS syntax for matching
- They allow very fast and powerful element matching, e.g.:

1. "#main" - returns the element with ID "main" – **by ID**

2. "#content div" - selects all **<div>**s inside #content – **by ID and tag name**

Връща с ID content и таг div

```
document.querySelectorAll('#outputs div p')[0].textContent = restaurantInfo; // Връща с ID
outputs и в таг div тага p – първият p с нулев индекс
```

```

<div id="outputs">
  <div id="bestRestaurant">
    <h2>Best Restaurant</h2>
    <span></span>
    <p></p>
  </div>
  <div id="workers">
    <h2>Best Restaurant's workers</h2>
    <span></span>
    <p></p>
  </div>

```

```
    </div>
</div>
```

3. ".note, .alert" - all elements with class "note" or "alert" – единият от двета класа е достатъчен – **by class name**

3.1. Първата точка е класа, останалите точки заместват space-a.

```
<div class = "card-deck d-flex justify-content-center"></div>
const catalog = section.querySelector('.card-deck.d-flex.justify-content-center');
```

3.2. Когато търсим по име на клас, може да използваме само част от името на класа. Пример:

```
<li class="nav-item1 user">
    <a id="welcomeMsg" class="nav-link">Welcome, email</a>
</li>
<li class="nav-item2 user">
    <a id="logoutBtn" class="nav-link" href="#">Logout</a>
</li>
```

```
[...nav.querySelectorAll('.user')].forEach(d => d.style.display = 'block');
[...nav.querySelectorAll('.guest')].forEach(d => d.style.display = 'none');
```

3.3. Tag name с даден клас

```
const root = document.querySelector('div.container'); //tag div с атрибут клас .container

<div class="container">
    <!--View content-->
</div>
```

3.3. Tag name и подобект с даден таг и клас

```
<body>
    <div id="content">
        <header id="titlebar" class="layout">
            <a href="#" class="site-logo">Team Manager</a>
            <nav>
                <a href="#" class="action">Browse Teams</a>
                <a href="#" class="action guest">Login</a>
                <a href="#" class="action guest">Register</a>
                <a href="#" class="action user">My Teams</a>
                <a href="#" class="action user">Logout</a>
            </nav>
        </header>
        <main>
```

```
//търси в таг nav елементите, всички елементи с таг a (anchor), които имат клас user/guest
if (userData) {
    [...document.querySelectorAll('nav > a.user')].forEach(a => a.style.display = 'block');
    [...document.querySelectorAll('nav > a.guest')].forEach(a => a.style.display = 'none');
```

```
    } else {
        [...document.querySelectorAll('nav > a.user')].forEach(a => a.style.display = 'none');
        [...document.querySelectorAll('nav > a.guest')].forEach(a => a.style.display = 'block');
    }
}
```

4. "input[name='login']" - <input> with name "login" – по параметър на html елемента (type, name, други)

```
const input = document.querySelector("input[name='email']");
```

```
let textTotalLikes = document.querySelector("div[class='likes'] p");
```

```
<label>
    Email: <input type="text" name="email" />
    <button onclick="deleteByEmail()">Delete</button>
</label>
```

Пример:

```
function onConvert(ev) {
    ev.target.parentElement.querySelector('input[type="text"]');
}
```

```
<div>
    <label for="days">Days: </label>
    <input type="text" id="days">
    <input id="daysBtn" type="button" value="Convert">
</div>
```

Пример с 2 параметъра на тага

```
const isActive = profile.querySelector('input[type="radio"][value="unlock"]').checked;
```

CSS Selectors - Example

- Select the **first matching** element

```
const mainDiv = document.querySelector('#main');
// # means select the element with ID 'main'
const element = document.querySelector('p');
// Select the first paragraph on the page
```

- Select **all** matching elements

- Returns a **static NodeList**

```
const elements = document.querySelectorAll('article.list');
//Select all <article> elements having a class named 'list'
```

```
const items = document.querySelectorAll('#items li');
//Select all elements having an id named 'items' and tag elements that have 'li'
```

NodeList vs. HTMLCollection

- Both interfaces are **collections of DOM nodes**
- **NodeList** can contain **any** node type, including **text** and **whitespace**
- **HTMLCollection** contains only **Element nodes**
- Both have **iteration** methods, **HTMLCollection** has an extra **namedItem** method
- **HTMLCollection** is **live**, while **NodeList** can be either **live** or **static**

Iterating Element Collections

- **NodeList** and **HTMLCollection** are **NOT arrays** but can be **indexed** and **iterated**

```
const elements = document.querySelectorAll('p');
const first = elements[0];
//Select the first paragraph on the page
for (let p of elements) { /* ... */ }
//Iterate over all entries
```

- Both can be **explicitly converted** to an array

```
const elementArray = Array.from(elements);
const elementArr2 = [...elements]; // Spread syntax
```

Задача 2:

```
function extractText() {
    const items = document.getElementById('items').children;//items id-то има в случая 3 деца
    const result = [];

    for (const item of items) {
        result.push(item.textContent);
    }

    document.getElementById('result').textContent = result.join('\n');
}

<!DOCTYPE html>
<html lang="en">
<head>
    <title>Collect List Items</title>
</head>
<body>
<ul id="items">
    <li>first item</li>
    <li>second item</li>
    <li>third item</li>
</ul>
<textarea id="result"></textarea>
<br>
<button onclick="extractText(); console.log("2nd JS code executing")">Extract Text</button>
// след onclick в кавичките е JS код.
<script src="CollectListItems.js">
</script>
```

```
</body>
</html>
```

Parents and Child Elements

- Every DOM Element has a **parent**
 - Parents can be accessed by property **parentElement** or **parentNode**

```
▼<div>
  <p>This is a paragraph.</p>
  <p>This is another paragraph.</p>
</div>
```

```
let firstP = document.getElementsByTagName('p')[0]; //Accessing the first child
console.log(firstP.parentElement); //Accessing the child's parent
```

```
►<div>...</div>
```

- When some element contains other elements, that means he is **parent** of those elements
- They are **children** to the **parent**. They can be accessed by property **children**

```
▼<div>
  <p>This is a paragraph.</p>
  <p>This is another paragraph.</p>
</div>
```

```
▼HTMLCollection(2) [p, p]
  ►0: p
  ►1: p
  length: 2
```

```
let pElements = document.getElementsByTagName('div')[0].children; //Returns live
HTMLCollection
```

14.5. Using the DOM API

External Page Scripts

- Page scripts can be **loaded** from an external file
 - Use the **src** attribute of the **script** element

```
<script src="app.js"></script>
```

- **Functions** from script files are in the **global scope**
 - Can be referenced and **executed** from **events** and **inline** scripts
 - **Multiple** script files in a page can see **each other**
 - Pay attention to **load order!**

Control Content via Visibility

- Content can be **hidden** or **revealed** by changing its **display** style
 - This is a **common technique** to display content dynamically

- To **hide** an element:

```
const element = document.getElementById('main');
element.style.display = 'none';
```

- To **reveal** an element, set **display** to anything that isn't '**none**' (including **empty string**)

```
element.style.display = ''; // Can be 'inline', 'block', etc.
```

Example:

```
function showText() {
    //select and reveal text
    const text = document.getElementById('text');
    text.style.display = 'inline';
    text.style.display = 'block';

    //select and hide button
    const btn = document.getElementById('more');
    btn.style.display = 'none';
}

<!DOCTYPE html>
<html lang="en">
<head>
    <title>Show More</title>
</head>
<body>
    Welcome to the "Show More Text Example".
    <a href="#" id="more" onclick= "showText()">Read more ...</a>
    <span id="text" style= "display:none">Welcome to JavaScript and DOM.</span>
    <script src="ShowMore.js"></script>
</body>
</html>
```

Match n-th Child

- Sometimes we need to target an element based on its **relation** to other **similar elements**
 - E.g., **row** or **column** in a table, **list item**, etc.
- Can be done either by **index** or with a **CSS selector**

```
const list = document.getElementsByTagName('ul')[0];
// First <ul> on the page
const thirdLi = list.getElementsByTagName('li')[2];
// Third <li> inside the selected <ul>

const thirdLi = document.querySelector('ul li:nth-child(3)');
// Third <li> inside the first <ul> on the page
```

Example:

```
function colorize() {
    const rows = document.querySelectorAll('table tr');
```

```

const rows = document.querySelectorAll('table tr:nth-child(even)');

for(let i = 1; i < rows.length; i+=2){
    rows[i].style.backgroundColor = 'teal';
}

}

function sumTable() {
    //select first table
    let rows = document.querySelectorAll('table tr');

    let sum = 0;

    //select only rows containing values
    // repeat for every row
    // - find last cell
    // - add cell value to sum
    for (let i = 1; i < rows.length - 1; i++) {
        const cell = Number(rows[i].lastElementChild.textContent); //последната колона
        sum += cell;
    }

    //select element with id "sum" and set its value
    document.getElementById('sum').textContent = sum;
}

```

lastElement може да хване whitespace – затова използваме **lastElementChild**
lastElementChild е наследник на NodeList и като такъв има повече екстри от **lastElement**.

На текущият елемент от html-а, през Global Variable си вземаме променлива, и на нея `.style` ни показва какви CSS стилове има този обект. И ние можем да ги променяме.

```
temp1.style.backgroundColor = "red";
```

14.6. Debuggin in the browser

Sources ->

Добавяне на Event listener

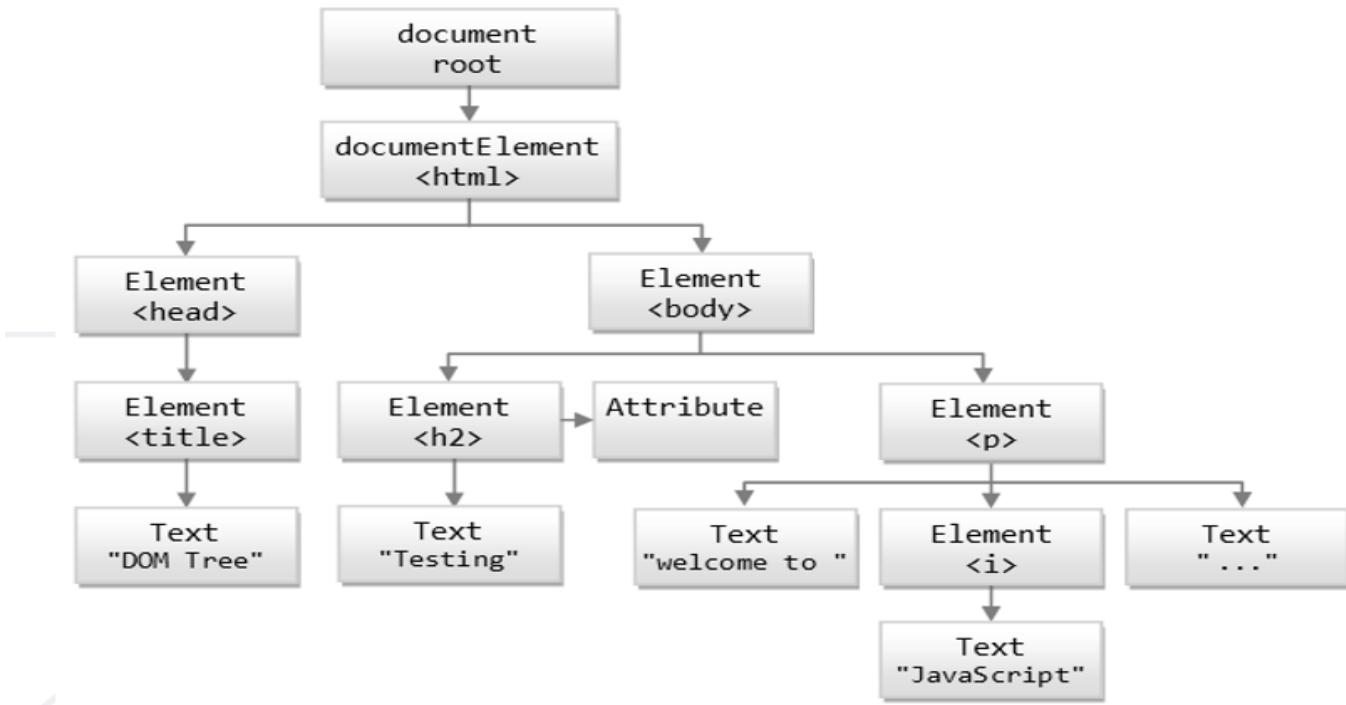
```
const button = document.querySelector('#searchBtn').addEventListener("click", onClick);
const button1 = document.getElementById('searchBtn').addEventListener("click", onClick);
```

Използване на text area

```
<textarea id="input" cols="30" rows="12"></textarea>
```

15. DOM Events

Handling DOM Events, Propagation & Delegation



15.1. DOM Manipulations

- We can **create**, **append** and **remove** HTML elements dynamically
 - `appendChild()`
 - `removeChild()`
 - `replaceChild()`

15.1.1. Creating New DOM Elements

- HTML elements are created with `document.createElement`
 - This is called a **Factory Pattern**
- Variables holding HTML elements are **live**:
 - If you **modify** the contents of the variable, the DOM is **updated**
 - If you **insert** it somewhere in the DOM, the original is **moved**
- Text added to `textContent` will be **escaped** – **няма да бъде интерпретирано**

В HTML има 3 специални символа, отварящ таг <, затварящ таг>, и амперсант &. Те могат да бъдат escape-нати чрез

```
function escapeHtml(value) {  
    return value  
        .toString()  
        .replace(/&/g, '&amp;')  
        .replace(/</g, '&lt;')  
        .replace(/>/g, '&gt;')  
        .replace(/\"/g, '&quot;')  
        .replace(/\'/g, '&#39;')  
}
```

- Text added to `innerHTML` will be **parsed** and turned into actual HTML elements → beware of **XSS attacks!**

Creating a new DOM element

```
let p = document.createElement("p"); //tag <p>
let li = document.createElement("li"); //tag <li>
```

Пример за hyperlink reference

```
let anchor = document.createElement("a"); //tag <a> </a>      href="#"
```

```
//create Delete button
const button = document.createElement("a");
button.textContent = '[Delete]';
button.href = '#';
button.addEventListener('click', removeElement);

function removeElement(event) {
    const li = event.target.parentNode;
    console.log(li.remove());
}
```

Creating manually innerHTML – с тилда кавички

```
movie.innerHTML = `<span>${name.value}</span>
<strong>${hall.value}</strong>
<div>
    <strong>${Number(ticketPrice.value).toFixed(2)}</strong>
    <input placeholder="Tickets sold">
    <button>Archive</button>
</div>`;
```

Create a copy / cloning DOM element

```
let li = document.getElementById("my-list");
let newLi = li.cloneNode(true);
```

- Elements are created **in memory** – they don't exist on the page
- To become visible, they must be **appended** to the DOM tree

Creating a new fragment element

DocumentFragments are DOM [Node](#) objects which are never part of the main DOM tree. The usual use case is to create the document fragment, append elements to the document fragment and then append the document fragment to the DOM tree. In the DOM tree, the document fragment is replaced by all its children.

Since the document fragment is *in memory* and not part of the main DOM tree, appending children to it does not cause page [reflow](#) (computation of element's position and geometry). Historically, using document fragments could result in [better performance](#)

```

function createIdeaDiv(idea) {
    const fragment = document.createDocumentFragment();

    fragment.appendChild(e('img', {className: 'det-img', src: idea.img}));
    fragment.appendChild(e('div', {className: 'desc'}),
        e('h2', {className: 'display-5'}, idea.title),
        e('p', {className: 'infoType'}, 'Description:'),
        e('p', {className: 'idea-description'}, idea.description)
    );
    fragment.appendChild(e('div', {className: 'text-center'}),
        e('a', {className: 'btn detb', href: ""}, 'Delete')
    );

    return fragment;
}

```

15.1.2. Manipulating Node Hierarchy

- **appendChild** - Adds a new child, as the **last child**. If the child is part of the list, it is firstly removed at the initial position, and then added as the last child

```

let p = document.createElement("p");
let li = document.createElement("li");
li.appendChild(p);

```

- **prepend** - Adds a new child, as the **first child**

```

let ul = document.getElementById("my-list");
let li = document.createElement("li");
ul.prepend(li);

```

15.1.3. Deleting DOM Elements

Изтритият елемент се маха от DOM дървото, но си остава в паметта на браузъра като референция

```

<ul id="items">
    <li class="red">Red</li>
    <li class="blue">Blue</li>
</ul>

let redElements = document.querySelectorAll("#items li.red");
redElements.forEach(li => li.parentNode.removeChild(li));

```

remove() - има и такъв метод, премахва текущия елемент

15.1.4. Replace Node from one list of elements to another list of elements

Първи начин:

```

const liElement = document.querySelector('ul').children[1]; //от първия списък
liElement.remove();

```

```
document.querySelectorAll('ul')[1].append(liElement); //втория списък
```

Втори начин:

Като добавим дадения DOM обект на ново място, то той автоматично се откача от първото място!!!

timeTableElement.replaceChildren(); - изтрива визуално съществуващите до момента node-ве.

```
list.replaceChildren();
Object.values(data).map(createItem).forEach(i => list.appendChild(i));
```

Може и така да се използва

```
list.replaceChildren(...Object.values(data).map(createItem));
```

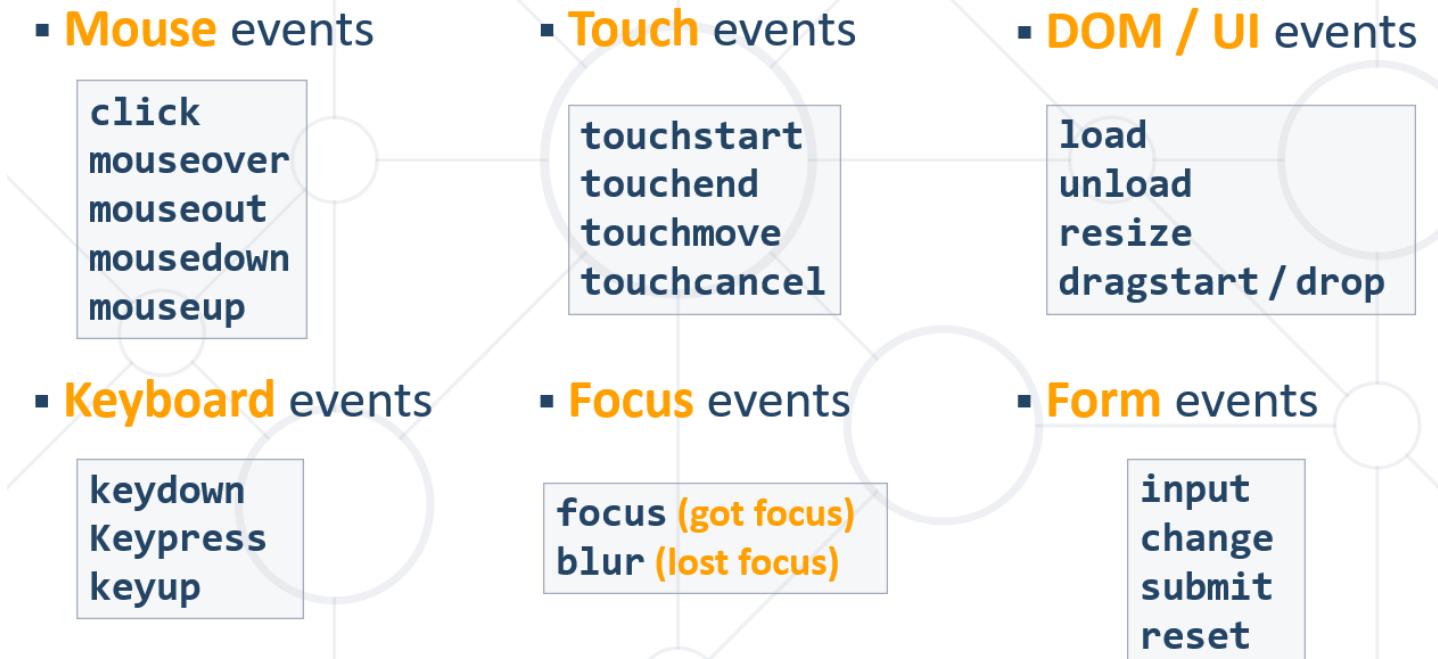
15.2. The DOM Event

Event Object and Types

Event Object

- Calls its **associated function**
- Passes a **single argument** to the function - a **reference** to the event object
- Contains **properties** that describe the event
 - Which **element** triggered the event
 - Screen **coordinates** where it occurred
 - What is the **type** of the event
 - And more

Event Types in DOM API



15.3. Event Handling

Event Handler = Listener

- Event registration is done by providing a **callback function**
- Three ways to register for an event:

1. With *HTML Attributes* – не е добре да се използва

Да го видя как става

2. Using *DOM element properties* – не е добре да се използва

```
<button id="btn">Click me</button>
<script>
    document.getElementById("btn").onclick = logEventData;

    function logEventData(event){
        console.log(event);
    }
</script>
```

3. Using *DOM event handler* – preferred method

```
function handler(event) {
    // this --> object, html reference
    // event --> object, event configuration
}

<body>
    <ul id="list">
        <li>Imalo</li>
        <li>Edno</li>
        <li>Vreme</li>
        <li>v Sofia</li>
    </ul>

    <button id="btn">Click me</button>

    <script>
        const btn = document.getElementById("btn");
        btn.addEventListener("click", logEventData); //вида на събитието, какво прави

        function logEventData() {
            console.log("Hello baby");
        }

        function logEventData(event) { //това ни дава самият event
            console.log(event);
        }

    </script>
</body>
```

```

Event Listener = Handler
addEventListener();
htmlRef.addEventListener( 'click' , handler , false );

const gradient = document.getElementById('gradient');
gradient.addEventListener('mousemove', onMouse);

removeEventListener();
htmlRef.removeEventListener( 'click' , handler);

gradient.removeEventListener('mousemove', onMouse);

```

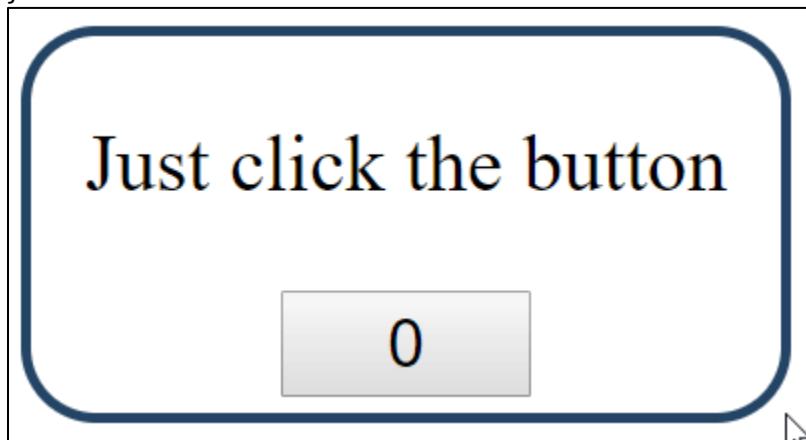
Събитие click

```

const button = document.getElementsByTagName('button')[0];
button.addEventListener('click', clickMe);

//можем да сложим на много бутони тази функция
function clickMe(e) { //браузърът го задава събитието e
    const target = e.currentTarget; // връща текущия HTML element
    const target = e.target; // връща HTML element, където се е появило събитието
    const targetText = target.textContent;
    target.textContent = Number(targetText) + 1;
}

```



Пример:

```

<td>
    <button class='edit'>Edit</button>
    <button class='delete'>Delete</button>
</td>

```

```

const tbody = document.querySelector('tbody');
tbody.addEventListener('click', onTableClick);

```

```

function onTableClick(event) {
  if (event.target.className == 'delete') {
    console.log('clicked delete button');
  } else if (event.target.className == 'edit') {
    console.log('clicked edit button');
  }
}
<td>
  <button class='edit'>Edit</button>
  <button class='delete'>Delete</button>
</td>

```

Events Handler Execution Context

- In event handlers, **this** refers to the event **source element == html element**
- ```

element.addEventListener("click", function (e) {
 console.log(this === e.currentTarget); //Always true
});

```
- Pay attention when using **object methods** as event listeners!
    - **this** may not behave as you expect with objects

При слагане на **this** в ламбда функция, друго се изпълнява!!!

**Не е добра практика да използваме this в Event handling!!!**

Събитие mouseover

```

const button = document.getElementsByTagName('div')[0];
button.addEventListener('mouseover', function (e) {
 const style = e.currentTarget;
 const { backgroundColor } = style;

 if (backgroundColor === 'white') {
 targetStyles.backgroundColor = '#234465';
 targetStyles.color = 'white';
 } else {
 targetStyles.backgroundColor = 'white';
 targetStyles.color = '#234465';
 }
});

```

Dialog confirmation box

```

function removeElement(event) {
 const answer = confirm('Are you sure');

 if (answer == true) {
 const li = event.target.parentNode;
 console.log(li.remove());
 }
}

```

```
}
```

This page says

Are you sure

OK

Cancel

Изпълнява се при зареждане на страницата още

```
<body onload="attachGradientEvents()">
</body>
```

Събитие focused и blur:

```
</div>
▼<div class="focused">
 <h1>Section 3</h1>
 <input type="text">
</div>
...
function focused() {
 const fields = Array.from(document.getElementsByTagName("input"));

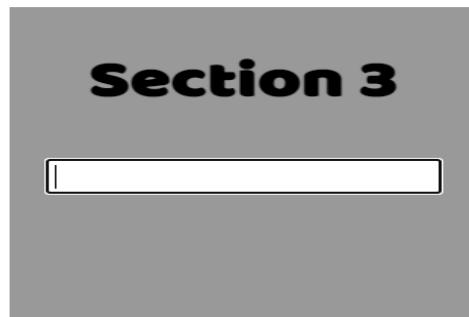
 for (const field of fields) {
 field.addEventListener('focus', onFocus);
 field.addEventListener('blur', onBlur);
 }

 function onFocus(ev) {
 ev.target.parentNode.className = 'focused';
 ev.target.parentNode.classList.add('focused'); //добавя клас на DOM елемента с име focused

 console.log(ev.target);
 }

 function onBlur(ev) {
 ev.target.parentNode.className = '';
 ev.target.parentNode.classList.remove('focused'); //премахва класа с име focused на DOM елемента

 console.log(ev.target);
 }
}
```



Има опция за използване на `classList` вместо `ev.target.parentNode.className = 'focused'`;

При `classList` е `contains` функцията, а не `includes`!

```
if (ev.target.tagName == 'BUTTON' && ev.target.classList.contains('add-product')) {
```

Събитие `change`

```
function validate() {
 document.getElementById("email").addEventListener('change', onChange); //излизаме от input-а и натискаме с мишката извън полето за въвеждане

 function onChange({ target }) { //деструктуриране
 console.log(target.value);
 const pattern = /^[a-z]+@[a-z]+\.[a-z]+$/;

 if (pattern.test(target.value)) {
 target.classList.remove('error');
 } else {
 target.classList.add('error');
 }
 }
}
```

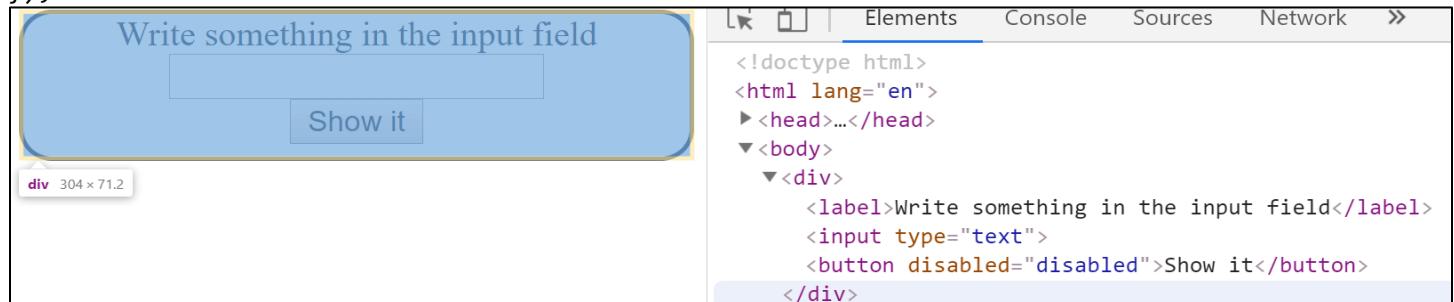
Събитие `onBlur`

Като `onChange`, но и без да има промяна в полето за въвеждане се активира събитието

Събитие `input` – при натискане на клавиш

```
const inputField = document.getElementsByTagName('input')[0];
const button = document.getElementsByTagName('button')[0];
```

```
inputField.addEventListener('input', function () {
 button.setAttribute('disabled', 'false')
});
```



Multiple Listeners

- The `addEventListener()` method also allows you to add many listeners to the same element, without overwriting existing ones:

```
element.addEventListener("click", myFirstFunction);
```

```
element.addEventListener("click", mySecondFunction);
element.addEventListener("mouseover", myThirdFunction);
element.addEventListener("mouseout", myFourthFunction);
```

Note that you don't use the "on" prefix for the event use "click" instead of "onclick"

## 15.4. Event Propagation

Handling Events Away From Their Source

Capture - Listener-a започва от най-глобалния елемент, и слиза да проверява DOM дървото надолу

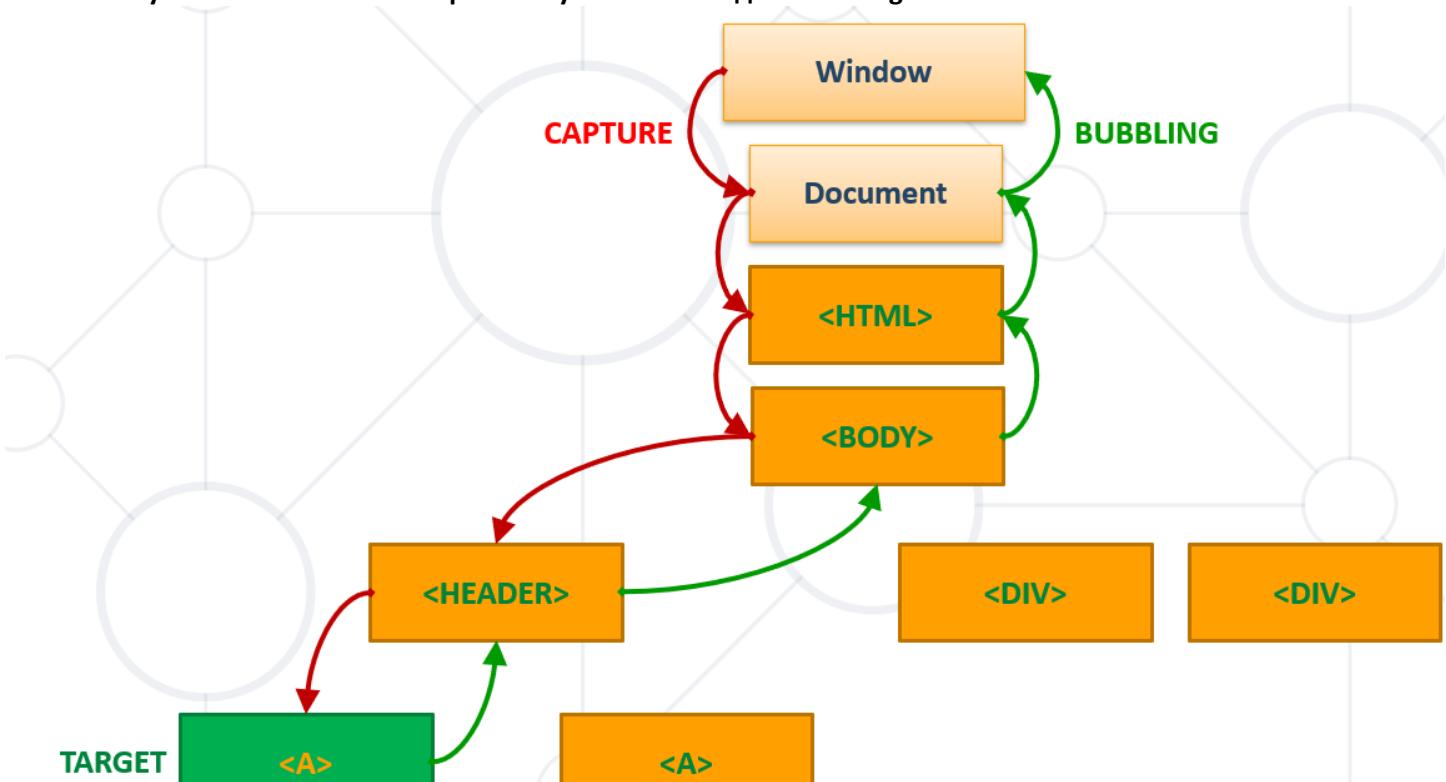
Bubbling – обратното

```
target.addEventListener(type, listener);
```

```
target.addEventListener(type, listener, useCapture);
```

- ако useCapture е true, то ще хванем capture фазата/модел на слушане на listener-а. Ако го заложим на false, ще се използва Bubbling фазата на слушане.

В 99% от случаите event Listener-а фазата му се използва да е Bubbling!



ev.currentTarget = евента от текущия елемент

ev.target = евента от същия тип, който ивент се е появил на първоначалния/съответния DOM HTML елемент

## DOM Event Delegation

- Allows you to **avoid** adding event listeners to specific nodes
- Event listener is assigned to a **single ancestor**

Пример 1:

```
<ul id="parent-list">
 <li id="post-1">Item 1
 <li id="post-2">Item 2
 Hyperlink 3

document.getElementById("parent-list")
 .addEventListener("click", function (e) {
 if (e.target && e.target.nodeName == "LI") { // e.target.nodeName == "A") anchor
 console.log(
 "List item ", e.target.id.replace("post-", ""),
 " was clicked!");
 }
 });
});
```

Пример 2:

```
document.querySelector('main').addEventListener('click', onConvert);

function onConvert(ev) {
 if (ev.target.tagName == 'INPUT' && ev.target.type == 'button') { //когато кликаме
 само на даден бутон(спрямо кода в HTML-а).
 const input = ev.target.parentElement.querySelector('input[type="text"]');
 const time = convert(Number(input.value), input.id);

 daysInput.value = time.days;
 hoursInput.value = time.hours;
 minutesInput.value = time.minutes;
 secondsInput.value = time.seconds;
 }
}
```

Пример 3: - когато работим с по-глобален listener, и той може да изпълнява различни listener-и

```
document.getElementById('main').addEventListener('click', onClick);
```

```
function onClick(e) {
 if (e.target.tagName == 'BUTTON') {
 onToggle(e);
 } else if (e.target.tagName == 'INPUT' && e.target.type == 'radio') {
 onLockToggle(e);
 }
}
```

Pros and Cons

- **Benefits**

- Simplifies initialization
- Saves memory

- Less code
- **Limitations**
  - Event must be bubbling
  - May add CPU load

## Controlling Propagation and Behavior

- **stopPropagation** prevents further propagation of the event – спира веригата
  - If there are **multiple handlers** for the same event
- **stopImmediatePropagation** prevents further propagation of the event – ако има няколко event listener-a закачени за даден html елемент, то като използваме **stopImmediatePropagation** освен че спира веригата, то и тези event-ти, които са на същия html елемент и те спират.
- **preventDefault** stop the browser from executing default behavior, for example:
  - **Navigating** to a new page when <a> is clicked – можем да спрем да ни се зарежда нова страница
  - Submitting **HTTP requests** via forms – можем да спрем формуларите да се самоизпращат
  - Opening **context menus** – можем когато потребителят щракне десен бутон с мишката, да не се отвори стандартното меню(за browse-ра), а наше си меню.

```
submitButton.addEventListener('click', (ev) => {
 ev.preventDefault();
 ...
});
```

15.5. Как правим в JS кода една променлива глобална – започваме от `window`. :

```
function convert(value, unit) {
 const inDays = value / ratios[unit];

 return {
 days: inDays,
 hours: inDays * ratios.hours,
 minutes: inDays * ratios.minutes,
 seconds: inDays * ratios.seconds,
 }
}

window.convert = convert;
```

Използване на `disabled` за бутон

```
//html
<button disabled="">Show more</button>
<input id="arrive" value="Arrive" type="button" onclick="result.arrive()" disabled="true">

//JS
const departBtn = document.getElementById('depart');
departBtn.disabled = true;
```

```
button:disabled{ //css
 background-color: #ccc;
}
```

## Form / Формуляр

Често се изпраща само към сървъра – например при кликане на бутона

Type	Description
<input type="text">	Displays a single-line text input field
<input type="radio">	Displays a radio button (for selecting one of many choices)
<input type="checkbox">	Displays a checkbox (for selecting zero or more of many choices)
<input type="submit">	Displays a submit button (for submitting the form)
<input type="button">	Displays a clickable button

### Формуляр с текст

```
<form id="add">
 <h2>Add New Pet</h2>
 <div id='container'>
 <input type="text" placeholder="Name" />
 <input type="text" placeholder="Age" />
 <input type="text" placeholder="Kind" />
 <input type="text" placeholder="Current Owner" />
 <button>Add</button>
 </div>
</form>
```

### Формуляр с радиобутон

```
<form>
 <input type="radio" name="sex" value="male">Мъж

 <input type="radio" name="sex" value="female">Жена
</form>
```

Формуляр с checkbox

```
<form>
 <input type="checkbox" name="vehicle" value="Bike">Имам мотор

 <input type="checkbox" name="vehicle" value="Car">Имам кола
</form>
```

Формуляр с submit бутон

```
<form action="/action_page.php">
 <label for="fname">First name:</label>

 <input type="text" id="fname" name="fname" value="John">

 <label for="lname">Last name:</label>

 <input type="text" id="lname" name="lname" value="Doe">

 <input type="submit" value="Submit">
</form>
```

## Add New Pet

```
Name _____ Age _____
Kind _____ Current Owner _____
Add
```

```
//get references to elements of interest
const fields = document.querySelectorAll('#container input');
const addBtn = document.querySelector('#container button');

//configure event listeners
addBtn.addEventListener("click", addPet);

function addPet(ev) {
 ev.preventDefault();
}
```

Алтернативен начин за изтриване на данни от формуляр

document.querySelector('form').reset(); - изтрива текста на всички input елементи

```
<form id="add">
 <h2>Add New Pet</h2>
 <div id='container'>
 <input type="text" placeholder="Name" />
 <input type="text" placeholder="Age" />
 <input type="text" placeholder="Kind" />
 <input type="text" placeholder="Current Owner" />
 <button>Add</button>
 </div>
</form>
```

## Радиобутон

Работи със свойство `.checked`

По `name` се засича селекцията от радиоизбори в случая

```
<div>
 <input type="radio" id="huey" name="drone" value="huey" checked>
 <label for="huey">Huey</label>
</div>

<div>
 <input type="radio" id="dewey" name="drone" value="dewey">
 <label for="dewey">Dewey</label>
</div>

<div>
 <input type="radio" id="louie" name="drone" value="louie">
 <label for="louie">Louie</label>
</div>
```

- Huey
- Dewey
- Louie

## Checkbox

`document.querySelectorAll('input[type="checkbox"]:checked');` - всички избрани checkbox-ве

```
<div>
 <input type="checkbox" id="horns" name="horns">
 <label for="horns">Horns</label>
</div>
```

```
document.getElementById("checkbox").addEventListener("change", (ev) => {
 if (ev.target.checked) {

 });
});
```

## Dropdown Menu

```
<select id="menu">
 <option value="1">Angular</option>
 <option value="2">React</option>
 <option value="3">Vue.js</option>
```

```

 <option value="4">Ember.js</option>
</select>

const element = document.createElement("option");
element.value = newValue;
element.textContent = newText;
const menu = document.querySelector("#menu").appendChild(element);

```

CSS border color

```

if (!passwordRegexPattern.test(passField.value)) {
 isValidInput = false;
 passField.style.borderColor = "red";
} else {
 userNameField.style.border = "none";
}

```

Creating img element and selecting source file

```

const imgElement = document.createElement("img");
imgElement.src = "./static/img/img.png";
divClassHitsInfo.appendChild(imgElement);

```

## 16. Advanced Functions

### 16.1. Execution Context

Execution Context Review

- The **function context** is the object that **owns** the currently executed code
- Function context === **this** object
- Depends on how the function is invoked
  - Global invoke: **func()**
  - Object method: **object.function()**
  - DOM Event: **element.addEventListener()**
  - Using **call()** / **apply()** / **bind()**

Inner Method Context and Global/Window context

- **this** variable is **accessible** only by the **outer method**

```

const obj = {
 name: 'Peter',
 outer() {
 console.log(this); // Object {name: "Peter"}
 function inner() { console.log(this); } // Global или Window
 inner();
 }
}

```

```
obj.outer(); // Window – в браузъра или Global ако е в Visual Studio Code/конзолата
```

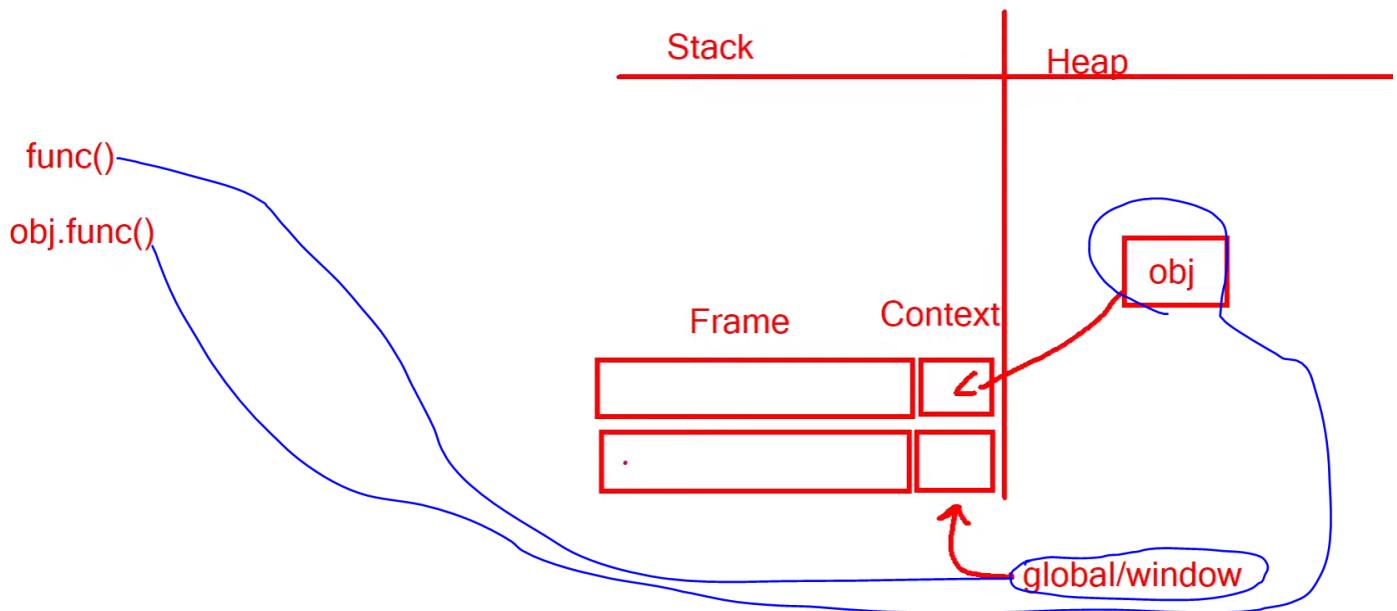
HTML DOM Element Context

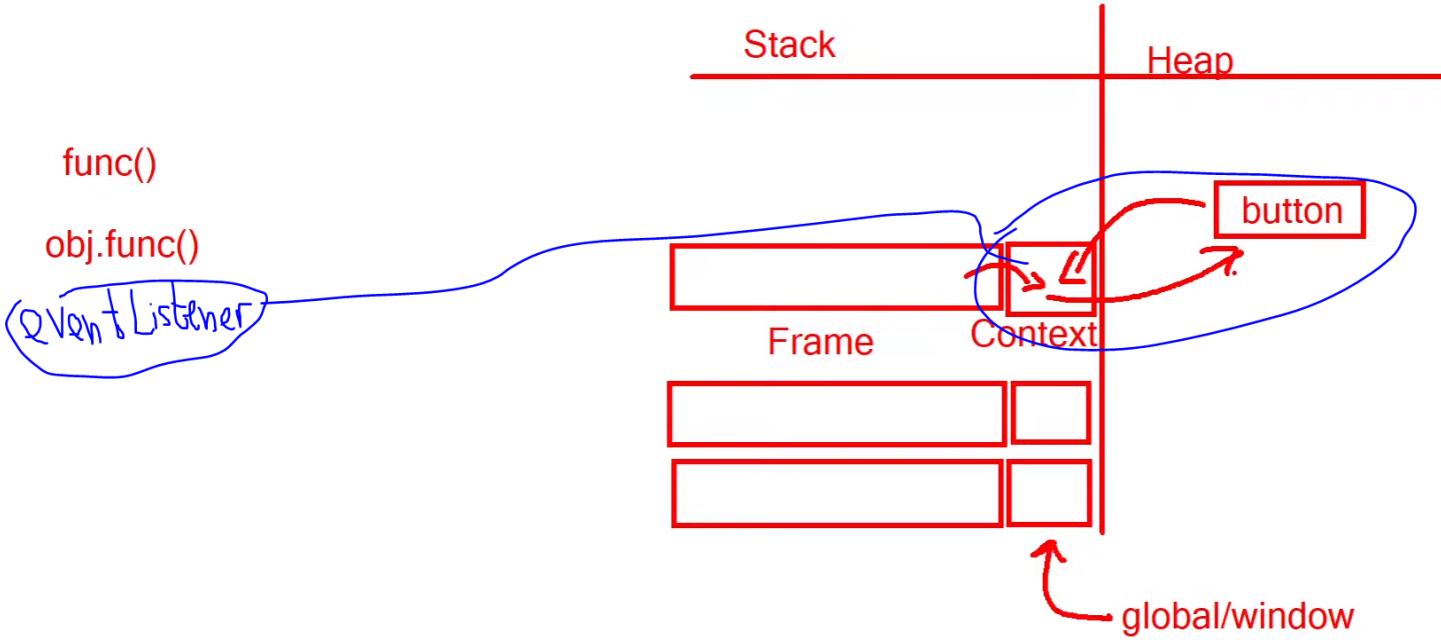
```
<button>Click me</button>
```

```
function contextPrinter() {
 console.log(this);
}
document.querySelector('button').addEventListener('click', contextPrinter); //this е html
бутоон елементът
```

Когато имаме myObj.contextPrinter, то се губи референцията към HTML DOM бутоонът/елементът

```
document.querySelector('button').addEventListener('click', myObj.contextPrinter);
```





### Arrow Function Context

- **this retains the value of the enclosing lexical context**
- Arrow функциите автоматично са bind-нати. Т.е. дори и на inner контекста е на обекта (на това което е било активно по време на декларацията), а не глобален
- **За разлика от function expressions, то Arrow-функциите запазват контекста на изпълнение, в който са деклариирани, независимо от къде се изпълняват**

```
const obj = {
 name: 'Peter',
 outer() {
 console.log(this); // Object {name: "Peter"}
 const inner = () => console.log(this); // Object {name: "Peter"}

 inner();
 }
}

obj.outer(); // Object {name: "Peter"}
```

### Explicit Binding – ние да можем да поставяме контекста

- Occurs when **call()**, **apply()**, or **bind()** are used on a function
- **Forces a function call to use a particular object for this binding**

```
function greet() {
 console.log(this.name);
}

let person = { name: 'Alex' };
```

```
greet.call(person, arg1, arg2, arg3, ...); // Alex
```

#### Changing the Context: Call

- Calls a function with a given **this** value and **arguments** provided individually

```
const sharePersonalInfo = function (...activities) {
 let info = `Hello, my name is ${this.name} and` + + `I'm a ${this.profession}.\\n`;
 info += activities.reduce((acc, curr) => {
 let el = `--- ${curr}\\n`;
 return acc + el;
 }, "My hobbies are:\\n").trim(); return info;
}
const firstPerson = { name: "Peter", profession: "Fisherman" };
console.log(sharePersonalInfo.call(firstPerson, 'biking', 'swimming', 'football'));
// Hello, my name is Peter.
// I'm a Fisherman.
// My hobbies are:
// --- biking
// --- swimming
// --- football
```

#### Друг пример за call

```
const obj = {
 id: 10,
 author: "Ivan",
 content: "AlaBalaNitsa",
 upvotes: 5,
 downvotes: 10,
}

function test() {
 this.upvotes += 8;
}

console.log(obj);
test.call(obj);
console.log(obj);
```

#### Changing the Context: Apply

- Calls a function with a given **this** value, and **arguments** provided as an **array**
- **apply()** accepts a **single array of arguments**, while **call()** accepts an **argument list**
- If the first argument is **undefined** or **null** a similar outcome can be achieved using the array **spread syntax**

```
const firstPerson = {
 name: "Peter",
 prof: "Fisherman",
 shareInfo: function () {
 console.log(`.${this.name} works as a ${this.prof}`);
 }
}
```

```

 }
};

const secondPerson = { name: "George", prof: "Manager" };
firstPerson.shareInfo.apply(secondPerson);
// George works as a Manager

```

**Пример – викаме на обекта cube функцията area – важи и когато обектът cube няма метод area**

```
const cubeArea = area.apply(cube); // --> cube.area()
```

*Changing the Context: Bind*

- The **bind()** method creates a **new function** – прави ново копие на функцията
- Has its **this** keyword **set** to the **provided** value, with a given sequence of arguments preceding any provided when the **new function** is called
- Calling the bound function generally results in the **execution** of its **wrapped function**
- Една функция можем да я преизползваме в/у колкото си искаем различни обекти и когато я байднем this ще сочи към съответния обект.

```

const x = 42;
const getX = function () {
 return this.x;
}
const module = { x, getX };
const unboundGetX = module.getX;
console.log(unboundGetX()); // undefined
const boundGetX = unboundGetX.bind(module);
console.log(boundGetX()); // 42

```

Object Methods as Browser Event Handlers

**Пример 1:**

```

▼ <body>
 <button id="callBtn">Call Person</button>
</body>

const person = {
 name: "Peter",
 respond() {
 alert(`${this.name} says hello!`); // Излиза диалогово съобщение
 }
}
const boundRespond = person.respond.bind(person);
document.getElementById('callBtn')
 .addEventListener('click', person.respond); // Unwanted result

document.getElementById('callBtn')
 .addEventListener('click', boundRespond); // Works as intended

```

## Пример 2:

```
confirmBtn.addEventListener('click', adopt.bind(null, confirmInput, pet)); //функцията adopt да се свързва с ниво на достъп null (към никой обект) и начални стойности confirmInput и pet - с цел да не се ровим из DOM дървото за искания обект.
```

Важно: След изброените bind-нати параметри, като скрит параметър е event(ev).

```
function adopt(input, pet) {
 const newOwner = input.value.trim();
 ..
}
```

Пример 3: Обектът event (ev) можем да го подадем като последен допълнителен елемент. Ако няма други елементи, то event (ev) се вика първи. В случая елементите, които са bind-нати са с предимство по поредност.

```
confirmBtn.addEventListener('click', adopt.bind(null, confirmInput, pet));
//функцията adopt да се свързва с ниво на достъп null (към никой обект) и начални стойности
confirmInput и pet
- с цел да не се ровим из DOM дървото за искания обект
```

```
confirmBtn.addEventListener('click', () => adopt(confirmInput, pet)); //също работи и с
ламбда функция
```

```
function adopt(input, pet, ev) {
 console.log(ev);
 const newOwner = input.value.trim();
 ..
}
```

## 16.2. Functional Programming in JS

### 16.2.1. First-Class Functions

- **First-class functions** are treated like any other variable
  - Passed as an **argument**
  - **Returned** by another function
  - Assigned as a **value** to a **variable**

The term "first-class" means that something is just a value. A first-class function is one that can go anywhere that any other value can go - there are few to no restrictions. - *Michael Fogus, Functional Javascript*

- Can be passed as an **argument** to another function

```
function sayHello() {
 return "Hello, ";
}

function greeting(helloMessage, name) {
 return helloMessage() + name;
}
```

```
console.log(greeting(sayHello, "JavaScript!"));
// Hello, JavaScript!
```

- Can be **returned** by another function
  - We can do that, because we treated functions in JavaScript as a **value**

```
function sayHello() {
 return function () {
 console.log('Hello!');
 }
}
```

- Can be assigned as a **value** to a **variable**

```
const write = function () {
 return "Hello, world!";
}

console.log(write());
// Hello, world!
```

#### 16.2.2. Higher-Order Functions – функции от по-висок ред

Или приема функция за параметър или връща нова функция като резултат.

- Take other **functions** as an **argument** or **return a function** as a result

```
const sayHello = function () {
 return function () {
 console.log("Hello!");
 }
}
```

```
const myFunc = sayHello();
myFunc(); // Hello!
```

#### 16.2.3. Predicates

- Any function that returns a **bool based** on evaluation of the **truth** of an **assertion**
- Predicates are often found in the form of **callbacks**

```
let found = array1.find(isFound); //isFound е предикат
```

```
function isFound(element) {
 return element > 10; //True or false
}

console.log(found); // 12
```

#### 16.2.4. Built-in Higher Order Functions

```
Array.prototype.map
Array.prototype.filter
Array.prototype.reduce
```

```

Array.prototype.find
Array.prototype.some
Array.prototype.every

users = [{ name: 'Tim', age: 25 },
{ name: 'Sam', age: 30 },
{ name: 'Bill', age: 20 }];

getName = (user) => user.name;
usernames = users.map(getName);
console.log(usernames) // ["Tim", "Sam", "Bill"]

```

#### 16.2.5. Pure Functions

- Returns the **same result** given **same parameters**
- Execution is **independent** of the state of the system
- Pure функцията не трябва да пипа нищо извън себе си

```

// impure function:
let number = 1; //външна променлива, външно състояние
const increment = () => number += 1;
console.log(increment()); // 2
console.log(increment()); // 3
console.log(increment()); // 4

// pure function:
const increment = n => n + 1;
console.log(increment(1)); // 2
console.log(increment(1)); // 2
console.log(increment(1)); // 2

```

#### 16.2.6. Referential Transparency

- An **expression** that can be **replaced** with its corresponding **value** without **changing** the program's behavior –  
свойството на една функция да може да бъде заменена с резултата от своето изпълнение
- Expression is **pure** and its evaluation must have no **side effects**

```

function add(a, b) { return a + b }; //чиста функция
function mult(a, b) { return a * b}; //чиста функция
let x = add(2, mult(3, 4));
// mult(3, 4) can be replaced with 12

```

#### 16.3. Closure - Inner Function State

- One of the most **important features** in JavaScript
- The **scope** of an inner function **includes** the scope of the outer function
- An **inner** function retains **variables** being used from the **outer** function scope even after the parent function has **returned**

**Вътрешният параметър counter на start() се променя/запомня при всяко извикване. Това се нарича closure – няма го в друг език това нещо!!!**

```

function start() {
 let counter = 0;
 function increment(a) {
 counter += a;
 console.log(counter);
 }

 return increment;
}

const myIncrement = start();
myIncrement(10);
myIncrement(1);
myIncrement(3);

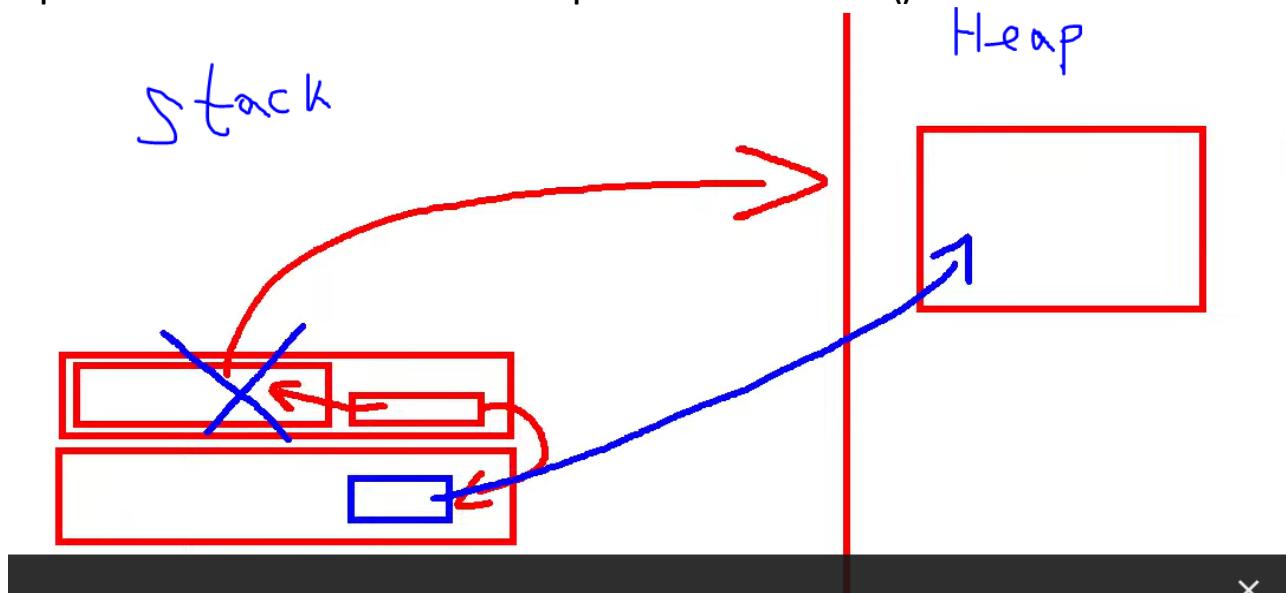
//10
//11
//14

const secondIncrement = start(); //викаме втори increment, но този път той започва отначало
secondIncrement(1);
secondIncrement(1);
secondIncrement(3);

//1
//2
//5

```

След приключване на първото изпълнение на функцията `start()`, променливите се пренасят в `heap`-а, откъдето се вземат наново от по-долния/следващия стек. Т.е. функцията `start()` и стойността на променливите ѝ от първото изпълнение са вече стойности от второто изпълнение на `start()`.



Друг пример за closure:

```
function getFibonator() {
```

```

let n1 = 0, n2 = 0;

function getNumber() {
 let nextNumber = n1 + n2;
 if (n1 == 0 && n2 == 0) {
 nextNumber += 1;
 }

 n1 = n2;
 n2 = nextNumber;

 return nextNumber;
}
return getNumber;
}

let fib = getFibonator();
console.log(fib()); // 1
console.log(fib()); // 1
console.log(fib()); // 2
console.log(fib()); // 3
console.log(fib()); // 5
console.log(fib()); // 8
console.log(fib()); // 13

```

### Още един пример за closure

```

function add(num) {
 let sum = num;

 function addInternal(num2) {
 sum += num2;
 return addInternal; //след като се е извикала веднъж, то се извиква и още един път!!
 }

 addInternal.toString = () => {
 return sum;
 }

 return addInternal;
}

// console.log(add(1).toString());
console.log(add(1)(6)(-3).toString()); //4 //функцията addInternal се извиква за 2-ри и 3ти
//елемент!!!

```

Енкапсулация без опция за Reflection

При енкапсулиране в JS не съществува сила на земята, която може да достъпи private полетата/променливите!

## Functions Returning Functions

- A **state** is preserved in the outer function (**closure**)

```
const f = (function () {
 let counter = 0;
 return function () {
 console.log(++counter);
 }
})();
```

What is IIFE?

- **Immediately-Invoked Function Expressions (IIFE)**
  - Define **anonymous** function expression
  - Invoke it **immediately** after declaration
  - Преди се е използвало доста, но сега вече имаме const и let

Пример 1:

```
(function () { let name = "Peter"; })(); //последните скоби изпълняват функцията, а самата
функция е в скоби
// Variable name is not accessible from the outside scope
console.log(name); // ReferenceError
```

Пример 2:

```
let result = (function () {
 let name = "Peter";
 return name;
})();
// Immediately creates the output:
console.log(result); // Peter
```

Пример 3: въпреки, че е мима, прототипът се задава за всички следващи инстанции

```
(function solve() {
 Array.prototype.last = function () {
 return this[this.length - 1];
 }

 Array.prototype.skip = function (n) {
 const result = [];
 for (let i = n; i < this.length; i++) {
 result.push(this[i]);
 }

 return result;
 }
})();
```

```
let arr = [1, 2, 3, 12];
console.log(arr.last());
console.log(arr.skip(2));
```

## 16.4. Function Decoration - Partial Application and Currying

### Partial Application

- Set some of the **arguments** of a function, **without executing it**
- Pass the **remaining arguments** when a result is needed
  - The partially applied function can be **used multiple times**
  - It will **retain all fixed arguments, regardless of context**

```
function pow(a, b) {
 return Math.pow(a, b);
}

const sqr = a => pow(a, 2);
console.log(sqr(2)); //4
console.log(sqr(3)); //9
console.log(sqr(4)); //16

function pow(power, num) {
 return Math.pow(num, power);
}

const sqr = pow.bind(null, 2);
console.log(sqr(2)); //4
console.log(sqr(3)); //9
console.log(sqr(4)); //16
```

### Currying

- Currying is a technique for **function decomposition**
- Supply arguments **one at a time**, instead of at once
  - They may come from **different sources**
  - Execution can be **delayed** until it's needed

#### Става възможно чрез closure-a

```
function sum3(a) {
 return (b) => {
 return (c) => {
 return a + b + c;
 }
 }
}
console.log(sum3(5)(6)(8)); // 19
```

- **Function Composition** - Building new function from old function by passing arguments
- **Memoization** - Functions that are called repeatedly with the same set of inputs but whose result is relatively expensive to produce
- **Handle Errors** - Throwing functions and exiting immediately after an error

## Currying vs Partial Application

- **Currying** always produces nested unary functions (функция с 1 параметър)
- **Partial** application produces functions of arbitrary number of arguments (с повече параметри)
- Currying is **NOT** partial application
  - But it can be implemented using partial application

# 17. Unit Testing and Error Handling

## 17.1. Error Handling

### Error Handling

- The fundamental **principle** of error handling says that a function (method) should either:
  - Do what its **name** suggests
  - Indicate a **problem**
  - Any other behavior is **incorrect**
- A function failed to do what its name suggests should:
  - Return a special value (e.g. **undefined / false / -1**)
  - Throw an **exception / error**
  - Exceptions are the **object-oriented way** for errors

```
let str = "Hello, SoftUni";
console.log(str.indexOf("Sofia")); // -1
// Special case returns a special value to indicate "not found"
```

### Types of Errors

- There are **three types** of errors in programming:
  - **Syntax Errors** - during parsing
  - **Runtime Errors** - occur during execution
    - After compilation, when the application is running
  - **Logical Errors** - occur when a mistake has been made in the logic of the script and the expected result is incorrect
    - Also known as bugs
- **Exception** - a function is unable to do its work (**fatal error**)

```
let arr = new Array(-1); // Uncaught RangeError
let bigArr = new Array(9999999999); // RangeError
let index = undefined.indexOf("hi"); // TypeError
console.log(George); // Uncaught ReferenceError
console.print('hi'); // Uncaught TypeError
```

- **Special Values**

```
let sqrt = Math.sqrt(-1); // NaN (special value)

let sub = "hello".substring(2, 1000); // llo
sub = "hello".substring(-100, 100); // hello
```

```
// Soft error - substring still does its job: takes all available chars

let invalid = new Date("Christmas"); // Invalid Date
let date = invalid.getDate(); // NaN
```

## Throwing Errors (Exceptions)

- The **throw** statement lets you create custom errors
  - **General Error** - `throw new Error ("Invalid state")`
  - **Range Error** - `throw new RangeError("Invalid index")`
  - **Type Error** - `throw new TypeError("String expected")`
  - **Reference Error** - `throw new ReferenceError("Missing age")`

```
throw new Error('Required');
// generates an error object with the message
```

## Try – Catch

- The **try** statement tests a block of code for **errors**
- The **catch** statement **handles** the error
- **Try** and **catch** come in pairs

```
try {
 // Code that can throw an exception
 // Some other code - not executed in case of error!
} catch (ex) {
 // This code is executed in case of exception
 // Ex holds the info about the exception
}
```

## Exception Properties and Finally

- An **Error object** with properties is created

```
try {
 throw new RangeError("Invalid range.");
 // console.log("This will not be executed.");
} catch (ex) {
 console.log("Exception object: " + ex);
 console.log("Type: " + ex.name);
 console.log("Message: " + ex.message);
 console.log("Stack: " + ex.stack);
} finally {
 console.log("I will execute every time");
}
```

!!Как прекратяваме изпълнението на mapping операцията, нали уж не можем да й спрем изпълнението

```
function createDeck(cards = []) {
functional solution
 try {
 console.log(cards.map(card => {
 const face = card.slice(0, -1);
```

```

 const suit = card.slice(-1);
 try {
 return createCard(face, suit);
 } catch (err) {
 throw new Error('Invalid card: ' + card);
 }
 }).join(" "));
} catch (error) {
 console.log(error.message);
}

```

Imperative solution

```

/*let result = [];
for (let card of cards) {
 const face = card.slice(0, -1);
 const suit = card.slice(-1);
 try {
 result.push(createCard(face, suit));
 } catch (err) {
 console.log('Invalid card: ' + card);
 return;
 }
}*/
```

```

// console.log(result.join(' '));

function createCard(face, suit) {
 const faces = ['2', '3', '4', '5', '6', '7', '8', '9', '10', 'J', 'Q', 'K', 'A'];
 const suits = {
 'S': '\u2660',
 'H': '\u2665',
 'D': '\u2666',
 'C': '\u2663',
 }

 if (faces.includes(face) == false) {
 throw new Error('Invalid face ' + face);
 }

 if (Object.keys(suits).includes(suit) == false) {
 throw new Error('Invalid suit ' + suit);
 }

 return {
 face,
 suit: suits[suit],
 toString() {
 return this.face + this.suit;
 }
 }
}

```

```
}
```

```
}
```

## 17.2. Unit Testing

### Unit Testing

- A **unit test** is a piece of code that checks whether certain functionality **works as expected**
- Allows developers to see **where & why errors occur**

```
function sortNums(arr) {
 arr.sort((a, b) => a - b);
}

let nums = [2, 15, -2, 4]; sortNums(nums);
if (JSON.stringify(nums) === "[-2,2,4,15]") {
 console.error("They are equal!");
}
```

Testing enables the following:

- **Easier maintenance** of the code base
  - Bugs are found ASAP
- **Faster development**
  - The so called "Test-driven development"
  - Tests before code
- **Automated way to find code wrongness**
  - If most of the features have tests, running them shows their correctness

### Unit Tests Structure

- The **AAA Pattern: Arrange, Act, Assert**

```
// Arrange all necessary preconditions and inputs
let nums = [2, 15, -2, 4];

// Act on the object or method under test
sortNums(nums);

// Assert that the obtained results are what we expect
if (JSON.stringify(nums) === "[-2,2,4,15]") {
 console.error("They are equal!");
}
```

### Unit Testing Frameworks

- JS Unit Testing:
  - [Mocha](#), [QUnit](#) (същото в JAVA е Junit; в C# е NUnit), [Unit.js](#), [Jasmine](#), [Jest \(All in one\)](#)
- Assertion frameworks (perform checks):
  - [Chai](#), [Assert.js](#), [Should.js](#)
- Mocking frameworks (mocks and stubs):
  - [Sinon](#), [JMock](#), [Mockito](#), [Moq](#)

## 17.3. Modules

### Modules

- Позволяват ни да отделим тестовете от главния код
- A **set of functions** to be included in applications
- Group related behavior
- Resolve naming collisions
  - `http.get(url)` and `students.get()`
- Expose only public behavior
  - They do not populate the global scope with unnecessary objects

a module for loading indicator

```
const loading = {
 show() {},
 hide() {},
};
```

### Node.js Modules

`require()` is used to *import* modules

```
const http = require('http');
// For NPM packages
```

```
const myModule = require('./myModule');
// For internal modules
```

```
const {sum} = require('./myModule'); //My written internal manual module - търси в локалната
директория
const {} = require('chai'); //importing library - без точка и наклонена черта - търси в
глобалния регистър node_modules
```

> node\_modules

- Internal modules need to be **exported before** being required
- In Node.js each file has its own scope

Whatever value has `module.exports` will be the value when using `require`

```
const myModule = () => {...};
module.exports = myModule;
```

```

JS symmetry.js > ...
1 function isSymmetric(arr) {
2 if (!Array.isArray(arr)){
3 return false; // Non-arrays are not symmetric
4 }
5 let reversed = arr.slice(0).reverse();
6 let equal = (JSON.stringify(arr) === JSON.stringify(reversed));
7 return equal;
8 }
9
10 module.exports = isSymmetric;

```

```

JS test.js > JS symmetry.js
JS test.js > ...
1 const { expect } = require('chai');
2 const isSymmetric = require('./symmetry.js');
3
4 describe('Symmetry Checker', () => {
5 it('return true for symmetric array', () => {
6 expect(isSymmetric([1, 2, 2, 1])).to.be.true;
7 });
8
9 it('returns false for non-symmetric array', () => {
10 expect(isSymmetric([1, 2, 3])).to.be.false;
11 })
12 })

```

Symmetry Checker

- ✓ return true for symmetric array
- ✓ returns false for non-symmetric array

2 passing (18ms)

To export more than one function, the value of `module.exports` will be an object

```
module.exports = { toCamelCase: convertToCamelCase, toLowerCase: convertToLowercase };
```

## 17.4. Unit Testing with Mocha and Chai

What is Mocha?

- Feature-rich JS test framework
- Provides common testing functions including `it`, `describe` and the `main` function that runs tests

```
describe("title", function () {
 it("title", function () { ... }); //it е отделен тест
});
```

- Usually used together with **Chai (за сравнение)**

What is Chai?

- A library with many assertions
- Allows the usage of a lot of different assertions such as `assert.equal`

```
let assert = require("chai").assert; //без деструктуриране
describe("pow", function () {
 it("2 raised to power 3 is 8", function () {
 assert.equal(pow(2, 3), 8);
 });
});
```

Друг пример:

```
const { expect } = require('chai');
const {expect, assert} = require("chai");
const { sum } = require('./myModule');

describe('Demo Test', () => {
 it('works with 5 and 3', () => {
 //Way with chai
 expect(sum(5, 3)).to.equal(9); //Вариант 1
 expect(sum(5, 3)).equal(8);
 expect(sum(5, 3)).to.be.equal(8);
 expect(sum(5, 3)).to.be.closeTo(8, 0.1); //близко до, с отклонение 0,1

 assert.equal(sum(5, 3), 8); //Вариант 2

 //way without chai - Вариант 3
 // if (sum(5, 3) != 8) {
 // throw new Error('Did not work as expected');
 // }
 });
}

AssertionError: expected 8 to equal 9
+ expected - actual

-8
+9

at Context.<anonymous> (test.js:8:20)
 Demo Test
 ✓ works with 5 and 3
```

1 passing (8ms)

Пример с вложени `describe` блокове:

```
const {expect} = require('chai');
const rgbToHexColor = require('./rgb-to-hex');

describe('RGB converter', () => {
 describe('Happy path', () => {
 it('converts white', () => {
 expect(rgbToHexColor(255, 255, 255)).to.equal('#FFFFFF')
 });

 it('converts black', () => {
 expect(rgbToHexColor(0, 0, 0)).to.equal('#000000')
 });

 it('converts SoftUni dark black', () => {
 expect(rgbToHexColor(35, 68, 101)).to.equal('#234465');
 });
 })
}

describe('Invalid parameter', () => {
 it('returns undefined for missing parameters', () => {
 expect(rgbToHexColor(255)).to.be.undefined;
 });

 it('returns undefined for values out of range', () => {
 expect(rgbToHexColor(-1, -1, -1)).to.be.undefined;
 });

 it('returns undefined for values out of range', () => {
 expect(rgbToHexColor(256, 256, 256)).to.be.undefined;
 });

 it('returns undefined for invalid parameter type', () => {
 expect(rgbToHexColor('5', 256, 'ass')).to.be.undefined;
 });
})
```

)

Пример със задаване на първоначално състояние

```
const { expect } = require('chai');
const { createCalculator } = require('./addSubtract.js');

describe('Summator', () => {
 let instance = null;

 beforeEach(() => {
 instance = createCalculator();
 }
```

```

);
it('Has all methods', () => {
 expect(instance).to.have.property('add');
 expect(instance).to.have.property('subtract');
 expect(instance).to.have.property('get');
});
it('Starts empty', () => {
 expect(instance.get()).to.equal(0);
});
it('Adds single number', () => {
 instance.add(1);
 expect(instance.get()).to.equal(1);
});

```

#### *Пример с викане/хващане на Exception*

Използваме/опаковаме с ламбда функция с скобички преди това

```

expect(() => rgbToHexColor()).to.throw();
expect(() => rgbToHexColor()).to.throw(Error);
expect(() => rgbToHexColor()).to.throw(RangeError); //типа грешка, която хвърля можем да му подадем
expect(() => new PaymentPackage('HR Services')).to.throw(Error, 'Value must be a non-negative number'); //съобщението на грешката можем да му подадем

```

```

it("The array with books is empty", () => {
 expect(() => library.findBook([], "Muhaha")).to.throw(Error, "No books currently available");
});

```

#### Installation

- To install **frameworks** and **libraries**, use the CMD
  - Installing **Mocha** and **Chai** through **npm**

**npm init -y** – да има отделна папка само за тестовете и без други задачи

**npm install -g mocha** global – на всеки проект го инсталираме

**npm install chai**

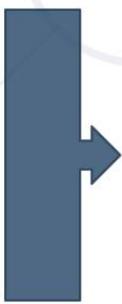
```

npm install --save chai
npm install --save mocha

```



```
npm init -y
npm install chai
npm install mocha
```



```
npm init -y
npm i chai mocha
```

```
{ } package-lock.json
{ } package.json
```

```
package.json
{
 "name": "testing",
 "version": "1.0.0",
 "description": "",
 "main": "index.js",
 "scripts": {
 "test": "node nameFile.test.js"
 },
 "keywords": [],
 "author": "",
 "license": "ISC",
 "dependencies": {
 "chai": "^4.3.4",
 "mocha": "^9.1.2"
 }
}
```

We run the test with the keyword **mocha** ако е глобално инсталирано

TERMINAL   DEBUG CONSOLE   PROBLEMS   OUTPUT

```
PS C:\Users\svilk\OneDrive\Soft Engineer\JAVA & JS path
t Testing and Error Handling - LAB\testing> mocha
Debugger listening on ws://127.0.0.1:59341/52b409d2-875
For help, see: https://nodejs.org/en/docs/inspector
Debugger attached.
```

Demo Test  
✓ passing test

1 passing (18ms)

Можем да извикаме/ръhnем и чрез **npm run test**, което от своя страна вика "test" от файлът **package.json**, в случая **mocha nameFile.test.js**

```
PS C:\Users\svilk\OneDrive\Soft Engineer\JAVA & JS path\3_Node JS core\3 - Advanced\real - September 2021\16 - Testing - EXC\Tes
g> npm run test
Debugger listening on ws://127.0.0.1:49632/1656d470-c45e-4d45-9a16-b25ef2a119f7
For help, see: https://nodejs.org/en/docs/inspector
Debugger attached.
```

```
> Testing@1.0.0 test C:\Users\svilk\OneDrive\Soft Engineer\JAVA & JS path\3_Node JS core\3 - Advanced\real - September 2021\16 -
sting - EXC\Testing
> mocha test.js
```

```
Debugger listening on ws://127.0.0.1:49637/f00364dc-b954-44c5-bc1b-a325cccd21f6f
For help, see: https://nodejs.org/en/docs/inspector
Debugger attached.
```

**0 passing (2ms)**

- To load a library, we need to **require** it

```
const expect = require("chai").expect; //именувани експорти/импорти, за да работи
IntelliSense-a
```

```
describe("Test group #1", function () {
 it("should... when...", function () {
 expect(actual).to.be.equal(expected);
 });
 it("should... when...", function () { ... });
});
describe("Test group #2", function () {
 it("should... when...", function () {
 expect(actual).to.be.equal(expected);
 });
});
```

## 17.5. How we run it in Judge

Пускаме само **describe** блока в Judge.

Също така е важно да следваме имената, които са дадени по условие.

### Как си пускаме различни mocha тестове от различни файлове

```
Waiting for the debugger to disconnect...
PS C:\Users\svilk\OneDrive\Soft Engineer\JAVA & JS path\3_Node JS core\3 - Advanced\real - September 2021\15 - Unit Testing and Err
or Handling - LAB\testing> mocha rgbToHexColor.test.js
```



```
{ package.json ×

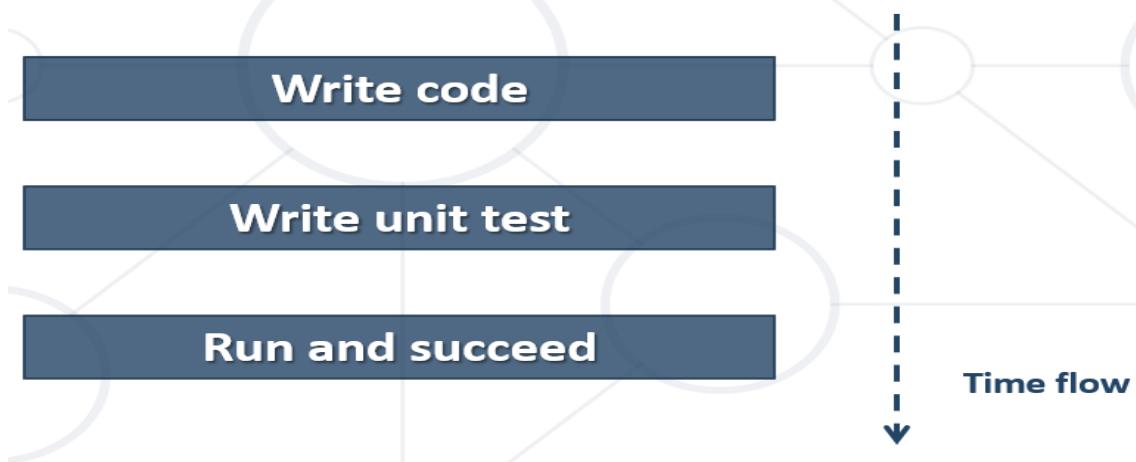
{} package.json > {} scripts
1 {
2 "name": "testing",
3 "version": "1.0.0",
4 "description": "",
5 "main": "index.js",
6 ▶ Debug
7 "scripts": [
8 "test": "mocha rgbToHexColor.test.js"
9],
10 "keywords": [],
11 "author": "",
12 "license": "ISC",
13 "dependencies": {
14 "chai": "^4.3.4",
15 "mocha": "^9.1.2"
16 }
}
```

Конвенция за именуване на тестовите файлове – името на основния файл с функция и след него слагаме **.test.js**

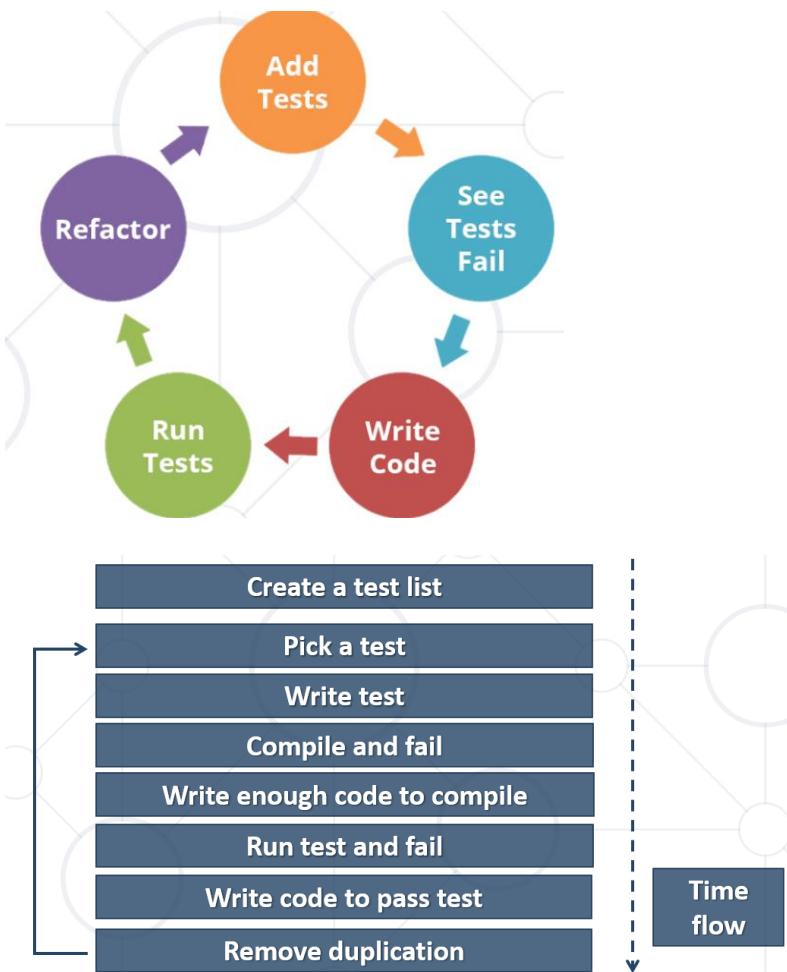
**JS** `rgbToHexColor.test.js`

## 17.5. Unit Testing Approaches

- **"Code First"** (code and test) approach - Classical approach



- **"Test First"** approach - Test-driven development (**TDD**)



## 18. JavaScript Classes

### 18.1. Defining Classes

#### Class Definition

- **Structure** for objects
- Classes define:
  - **Data** (properties, attributes)
  - **Actions** (behavior)
- One class may have **many instances** (objects)
- Unlike functions, class declarations are **not hoisted!** – Конвенцията е класът да се намира в отделен файл. Така че не е проблем да следим кое след кое сме декларирали.
- По конвенция, името на класа е **PascalCase**

#### Class Body

- The class body contains **method definitions**
- **Нямаме constructor overloading**

```
class Circle {
 constructor(r) {
 this.r = r;
 }
};
```

```

class Person {
 constructor(firstName, lastName, age, email) {
 // this.firstName = firstName;
 // this.lastName = lastName;
 // this.age = age;
 // this.email = email;
 Object.assign(this, { //вариант 2
 firstName,
 lastName,
 age,
 email,
 });
 }
}

```

- The **constructor** is a special method for **creating** and **initializing** an object created with a class
- Instance **properties** are defined inside the **constructor**

Служебно име на конструктора

Конструкторът има служебно име, което е името на класа като стринг

```

class Person {
 constructor(name, email) {
 this.name = name;
 this.email = email;
 }
 toString() {
 let className = this.constructor.name;
 console.log(className); //Person
 return `${className} (name: ${this.name}, email: ${this.email})`;
 }
}

```

## Class Methods

- A class may have **methods**, which will be available to its **instances**
- **Методите на класа реално не са част от инстанцията, а са закачени за неговия прототип**

```

class Rectangle {
 constructor(height, width) {
 this.height = height;
 this.width = width;
 }
 // Method
 calcArea() { return this.height * this.width; }
}
const square = new Rectangle(10, 10);
console.log(square.calcArea()); // 100

```

## Instance Context

- `this` refers to the **instance** of the class

```
class Person {
 constructor(firstName, lastName) {
 this.firstName = firstName;
 this.lastName = lastName;
 }
 displayName() {
 console.log(`Name: ${this.firstName} ${this.lastName}`);
 }
};
const person = new Person("John", "Doe");
person.displayName(); // Name: John Doe
```

Override на `toString()` метода

*Пример 1:*

```
class Person {
 constructor(firstName, lastName, age, email, secret) {

 this.firstName = firstName;
 this.lastName = lastName;
 this.age = age;
 this.email = email;

 this.secretField = () => secret; //привилегиран метод, до който има достъп само
closure-a
 }

 //Overriding of methods
 toString() {
 return `${this.firstName} ${this.lastName} (age: ${this.age}, email: ${this.email})`;
 }
};

const person = new Person("John", "Smith", 32, 'sss@abv.bg', 'My secret message');
console.log(person.toString()); // връща override-натия John Smith (age: 32, email: sss@abv.bg)
console.log('' + person); // връща override-натия John Smith (age: 32, email: sss@abv.bg)
console.log(`$ {person}`); // връща override-натия John Smith (age: 32, email: sss@abv.bg)

console.log(person); //принтира обекта - вика бащиния toString()
Person { firstName: 'John', lastName: 'Smith', age: 32, email: 'sss@abv.bg' }

console.log(person.secretField()); //My secret message
```

Пример 2:

```
function solution() {
 class Post {
 constructor(title, content) {
 this.title = title;
 this.content = content;
 }

 //overriding the prototype toString() method of class Post
 toString() {
 return `Post: ${this.title}\nContent: ${this.content}`;
 }
 }

 class SocialMediaPost extends Post {
 constructor(title, content, likes, dislikes) {
 super(title, content);
 this.likes = likes;
 this.dislikes = dislikes;
 this.comments = [];
 }

 addComment(comment) {
 this.comments.push(comment);
 }

 //overriding the toString() of class Post
 toString() {
 let result = super.toString() + `\nRating: ${this.likes - this.dislikes}\nComments:`;
 this.comments.forEach(comm => {
 result += `\n * ${comm}`;
 });

 return result;
 }
 }

 class BlogPost extends Post{
 constructor(title, content, views){
 super(title, content);
 this._views = views;
 }

 view() {
 this._views += 1;
 return this;
 } //instan.view().view().view();

 toString() {
```

```

 return super.toString() + `\\nViews: ${this._views}`;
 }
}

return { Post, SocialMediaPost}
}

const classes = solution();
let scm = new classes.SocialMediaPost("TestTitle", "TestContent", 25, 30);
scm.addComment("Good post");
scm.addComment("Very good post");
scm.addComment("Wow!");
console.log(scm.toString());

let blg = new classes.BlogPost("TestTitle", "TestContent", 10);
blg.view().view();
blg.view();
console.log(blg.toString());

```

Override на `valueOf()` метода

```
//Overriding of methods
valueOf() {
}
```

Instanceof Operator

- The `instanceof` operator returns `true` if the given object is an `instance` of the specified class

```
const circle = new Circle(5);
console.log(circle instanceof Circle); // true
console.log(circle instanceof Object); // true
console.log(circle instanceof String); // false
console.log(circle instanceof Number); // false
```

Static Methods

- The `static` keyword defines a `static method` for a class

```
class MyClass {
 static staticMethod() { return 'Static call'; }
}
```

- Static methods are `part of the class` and not of its instances

```
console.log(MyClass.staticMethod());
```

- They can **only** access other static methods via **this** context – статичните неща имат достъп само до други статични неща

```
static anotherStaticMethod() {
 return this.staticMethod() + ' from another method'; //ако имаме анонимен клас, то
MyClass.staticMethod() не можем да го извикаме, и използваме this.staticMethod()
}
```

### Static Field/parameter

```
class Person {
 static id = 0;
}
console.log(Person.id); // 0
```

### New syntax - field/parameter – private/public

```
class Person {
 #privateFiledID = 0;

 getPrivateField() {
 return this.#privateFiledID;
 }
}

console.log(Person.privateFiledID); //undefined
console.log(person.getPrivateField());
```

### Accessor Properties

- Accessor properties are **methods** that **mimic values**
  - Keywords **get** and **set** with **matching identifiers**
  - They can be **accessed** and **assigned** to like properties

```
class Circle {
 constructor(r) { this.radius = r; }
 get area() {
 return Math.PI * (this.radius ** 2);
 }
}
const circle = new Circle(5);
console.log(circle.area); //Accessing value without brackets - 78.5398...
```

### Пример:

```
class Circle {
 constructor(radius) { this.radius = radius; }
 get diameter() { return 2 * this.radius; } //Property getter
 set diameter(value) { //Property setter
 this.radius = value / 2;
```

```

}

get area() { //Read-only property area
 return Math.PI * (this.radius ** 2);
}

let c = new Circle(2);
c.diameter = 1.6; //използване на Сетъра!!!
console.log(`Radius: ${c.radius}`); // 0.8
console.log(`Diameter: ${c.diameter}`); // 1.6
console.log(`Area: ${c.area}`); // 2.0106...

```

## Accessor Properties Application

- Accessors are often used for **validation**
  - The **setter** can verify that a **given value** meets requirements
- Properties **without** a setter are **read-only** (cannot be assigned) – има само getter
- Getters can be used for a **validated** or **calculated** property

### Пример 1

```

class PaymentPackage {
 constructor(name, value) {
 this.name = name; //вика сетъра set name(newValue == name)
 this.value = value; //вика сетъра set value(newValue == value)
 this.VAT = 20; // Default value //вика сетъра set VAT(newValue == 20)
 this.active = true; // Default value //вика сетъра set active(newValue == true)
 }

 get name() {
 return this._name; //връща създаденото мнимо по конвенция поле _name - то не е
private, но го смятаме/различаваме като private
 }

 set name(newValue) {
 if (typeof newValue !== 'string') {
 throw new Error('Name must be a non-empty string');
 }
 if (newValue.length === 0) {
 throw new Error('Name must be a non-empty string');
 }
 this._name = newValue; //създава по конвенция мнимо поле _name - то не е private, но
го смятаме/различаваме като private
 }

 get value() {
 return this._value;
 }
}

```

```

set value(newValue) {
 if (typeof newValue !== 'number') {
 throw new Error('Value must be a non-negative number');
 }
 if (newValue < 0) {
 throw new Error('Value must be a non-negative number');
 }
 this._value = newValue; //създава по конвенция мимо поле _value - то не е private,
но го смятаме/различаваме като private
}

get VAT() {
 return this._VAT;
}

set VAT(newValue) {
 if (typeof newValue !== 'number') {
 throw new Error('VAT must be a non-negative number');
 }
 if (newValue < 0) {
 throw new Error('VAT must be a non-negative number');
 }
 this._VAT = newValue; //създава по конвенция мимо поле _VAT - то не е private, но го
смятаме/различаваме като private
}

get active() {
 return this._active;
}

set active(newValue) {
 if (typeof newValue !== 'boolean') {
 throw new Error('Active status must be a boolean');
 }
 this._active = newValue; //създава по конвенция мимо поле _active - то не е private,
но го смятаме/различаваме като private
}

toString() {
 const output = [
 `Package: ${this.name}` + (this.active === false ? ' (inactive)' : ''),
 `- Value (excl. VAT): ${this.value}`,
 `- Value (VAT ${this.VAT}[]): ${this.value * (1 + this.VAT / 100)}`
];
 return output.join('\n');
}

```

```

}

// Should throw an error
try {
 const hrPack = new PaymentPackage('HR Services');
} catch(err) {
 console.log('Error: ' + err.message);
}

const packages = [
 new PaymentPackage('HR Services', 1500),
 new PaymentPackage('Consultation', 800),
 new PaymentPackage('Partnership Fee', 7000),
];
console.log(packages.join('\n'));

const wrongPack = new PaymentPackage('Transfer Fee', 100);
// Should throw an error
try {
 wrongPack.active = null;
} catch(err) {
 console.log('Error: ' + err.message);
}

```

*Пример 2:*

```

function solution() {
 class Balloon {
 constructor(color, hasWeight) {
 this.color = color;
 this.hasWeight = hasWeight;
 }
 }

 class PartyBalloon extends Balloon {
 constructor(color, hasWeight, ribbonColor, ribbonLength) {
 super(color, hasWeight);
 this._ribbon = { //създаваме без setter - то не е private, но го
 смитаме/различаваме като private

 color: ribbonColor,
 length: ribbonLength,
 }
 }

 get ribbon() {
 return this._ribbon;
 }
 }
}

```

```

}

class BirthdayBalloon extends PartyBalloon {
 constructor(color, hasWeight, ribbonColor, ribbonLength, text) {
 super(color, hasWeight, ribbonColor, ribbonLength);
 this._text = text; //създаваме без setter - то не е private, но го
смятаме/различаваме като private
 }

 get text(){
 return this._text;
 }
}

return {
 Balloon: Balloon,
 PartyBalloon: PartyBalloon,
 BirthdayBalloon: BirthdayBalloon,
}
}

let classes = solution();
let testBalloon = new classes.Balloon("Tumno-bqlo", 20.5);
console.log(testBalloon);

let testPartyBalloon = new classes.PartyBalloon("Tumno-bqlo", 20.5, "Svetlo-cherno", 10.25);
console.log(testPartyBalloon);

let ribbon = testPartyBalloon.ribbon;
console.log(ribbon);

let testBirthdayBalloon = new classes.BirthdayBalloon("Tumno-bqlo", 20.5, "Svetlo-cherno",
10.25, "Some text here");
console.log(testBirthdayBalloon);

```

## 18.2. DOM Classes

### DOM Elements as Class Instances

- All DOM objects are **instances** of standard DOM classes
  - Always created via **factory functions**, instead of with **new**

```
const divElement = document.createElement('div');
console.log(divElement instanceof HTMLDivElement); // true
```

- They provide many useful methods and properties
  - Already seen: **addEventListener**, **appendChild**, **remove**, **children**, **parentNode**, **textContent**, **value**, etc.

```
const div2 = new HTMLDivElement(); // класът HTMLDivElement е абстрактен, и не можем да създаваме инстанция от него
```

## Additional DOM Methods

- **cloneNode(deep)** create a **duplicate** of the selected element

- If **deep** is true, a **deep-copy** is created

```
const duplicate = divElement.cloneNode(true);
```

- **replaceWith()** replaces selected element with another

```
const span = document.createElement('span');
divElement.replaceWith(span);
```

- **before()** insert element before selected node

- **after()** insert element after selected node

## Manipulate Element CSS Class

- **classList** - is a read-only property that returns a collection of the class attributes of specified element

```
<div id="myDiv" class="container div root"></div>
```

```
const element = document.getElementById('myDiv').classList;
// DOMTokenList(3)
//["container", "div", "root", value: "container div root"]
```

- **classList Methods**

```
<div id="myDiv" class="container div root"></div>
```

- **add()** - Adds the specified class values

```
document.getElementById('myDiv').classList.add('testClass');
```

- **remove()** - Removes the specified class values

```
document.getElementById('myDiv').classList.remove('container');
```

```
<div id="myDiv" class="div root testClass"></div>
```

## HTML Attributes and Methods

- **getAttribute()** - returns the value of attributes of specified HTML element

```
<input type="text" name="username"/>
```

```
<input type="password" name="password"/>
```

```
const inputEle = document.getElementsByTagName('input')[0];
```

```
inputEle.getAttribute('type'); // text
```

```
inputEle.getAttribute('name'); // username
```

- **setAttribute()** - sets the value of an attribute on the specified HTML element

```
<input type="text" name="username"/>
<input type="password"/>

const inputPassEle = document.getElementsByTagName('input')[1];
inputPassEle.setAttribute('name', 'password');

<input type="text" name="username"/>
<input type="password" name="password"/>
```

**Пример за динамично създаване на елементи:**

```
const divClassHitsInfo = document.createElement("div");
divClassHitsInfo.setAttribute("class", "hits-info"); //създай клас с име hits-info
whereWeAddTheSongs.appendChild(divClassHitsInfo);

▼<div class="hits-info">

 <h2>Genre: Pop</h2>
</div>
```

- **removeAttribute()** - removes the attribute with the specified name from an HTML element

```
<input type="text" name="username" placeholder="Username..."/>
<input type="password" name="password" placeholder="Password..."/>

const inputPassEle = document.getElementsByTagName('input')[1];
inputPassEle.removeAttribute('placeholder');

<input type="text" name="username" placeholder="Username..."/>
<input type="password" name="password"/>
```

- **hasAttribute()** - method returns true if the specified attribute exists, otherwise it returns false

```
<input type="text" name="username" placeholder="Username..."/>
<input type="password" name="password" id="password"/>

const passwordElement = document.getElementById('password');
passwordElement.hasAttribute('name'); // true
passwordElement.hasAttribute('placeholder'); // false
```

- **dataset** obtain **DOMStringMap** of custom **data attributes**

```
function onDelete(event) {
 const id = event.target.dataset.id; // dataset има всеки елемент, но само бутона има и
 елементът id
```

## Combining Elements and Behavior

- Classes can be used to **encapsulate** elements and behavior

- Store **references** to DOM elements
- Provide **event handlers**
- **Methods** that **manipulate** the elements
- This is called the **Component Pattern**
  - Used in many **JS frameworks**, such as React, Vue, Angular
  - Used in the **Custom Web Component API** – идва наготово и работи на всеки браузър

Създаваме клас, който един вид декларира някакъв HTML елемент.

Разлика между HTML атрибути и DOM свойства

#### При използване на HTML атрибути

```
<input type="text" class="my-class" name="username" placeholder="Username..."/>
<input type="password" class="my-class" name="password" placeholder="Password..."/>
```

```
const inputPassEle = document.getElementsByTagName('input')[1];
inputPassEle.removeAttribute('placeholder');
inputPassEle.removeAttribute('class'); //тук изтриваме класа като атрибут
```

```
<input type="text" class="my-class" name="username" placeholder="Username..."/>
<input type="password" name="password"/>
```

#### При използване на DOM свойства:

```
<input type="text" class="my-class" name="username" placeholder="Username..."/>
<input type="password" class="my-class" name="password" placeholder="Password..."/>
const inputPassEle = document.getElementsByTagName('input')[1].classList;
console.log(inputPassEle);
inputPassEle.remove('my-class'); //тук класа си остава със стойност ""
```

```
<input type="text" class="my-class" name="username" placeholder="Username..."/>
<input type="password" class name="password" placeholder="Password..."/>
```

Пример за използване на клас в DOM

```
class Contact {
 constructor(firstName, lastName, phone, email) {
 this.firstName = firstName;
 this.lastName = lastName;
 this.phone = phone;
 this.email = email;
 this._online = false; //Полето с долната черта да не бъде достъпно извън класа
 }

 get online() {
 return this._online;
 }
}
```

```

set online(value) {
 if (this.titleDiv) { //Ако съществува
 if (value == false) {
 this.titleDiv.classList.remove("online");
 } else {
 this.titleDiv.classList.add("online");
 }
 }

 this._online = value;
}

render(id) {
 const article = document.createElement("article");

 this.titleDiv = document.createElement("div");
 this.titleDiv.classList.add("title");
 this.titleButton = document.createElement("button");
 this.titleButton.innerHTML = "ℹ";
 this.titleDiv.innerHTML = `${this.firstName} ${this.lastName}`;
 this.titleDiv.appendChild(this.titleButton);

 if (this._online) {
 this.titleDiv.classList.add("online");
 }

 this.infoDiv = document.createElement("div");
 this.infoDiv.classList.add("info");
 this.infoDiv.style.display = "none";
 this.infoDiv.innerHTML = `☎ ${this.phone}✉
${this.email}`;

 this.titleButton.addEventListener("click", () => {
 this.infoDiv.style.display = this.infoDiv.style.display == "none" ? "block" :
"none";
 });
}

article.appendChild(this.titleDiv);
article.appendChild(this.infoDiv);

document.getElementById(id).appendChild(article);
}

let contacts = [
 new Contact("Ivan", "Ivanov", "0888 123 456", "i.ivanov@gmail.com"),
 new Contact("Maria", "Petrova", "0899 987 654", "mar4eto@abv.bg"),
 new Contact("Jordan", "Kirov", "0988 456 789", "jordk@gmail.com")
];

```

```

contacts.forEach(c => c.render('main'));

// After 1 second, change the online status to true
setTimeout(() => contacts[1].online = true, 2000); //setter задава стойност true

```

## 18.3. Build-in Collections - Maps

### Set, Map, WeakSet, WeakMap

What is a Map?

- A **Map** object stores its elements in **insertion order** - за разлика, при обектите реда на вкарване може да е различен!!!
- A for-of loop returns an array of **[key, value]** for each iteration
- Pure **JavaScript objects** are like **Maps** in that both let you:
  - Assign **values** to **keys**
  - Detect whether something is stored in a key
  - Delete keys

**При обектите, задължително ключът е string. Докато при Map може ключът да е всякакъв вид данни**

Adding/Accessing Elements

*.set(key, value) – adds a new key-value pair*

```

let map = new Map();
map.set(1, "one");
map.set(2, "two");

```

*.get(key) – returns the value of the given key*

```

map.get(2); // two
map.get(1); // one

```

*.size – property, holding the number of stored entries*

```
map.size();
```

*.has(key) - checks if the map has the given key*

```

map.has(2); // true
map.has(4); // false

```

*.delete(key) - removes a key-value pair*

```
map.delete(1); // Removes 1 from the map
```

*.clear() - removes all key-value pairs*

```
map.clear();
```

Iterators

- **.entries()** - returns Iterator (докато при обектите връща масив) - array of **[key, value]** –
- **.keys()** - returns Iterator (докато при обектите връща масив) with all the **keys**

- `.values()` - returns Iterator(докато при обектите връща масив) with all the values

```
let entries = Array.from(map.entries()); // [[1, 'one'], [2, 'two']] - поредицата от данни
ги правим на масив
let keys = Array.from(map.keys()); // [1, 2]
let values = Array.from(map.values()); // ['one', 'two']
```

## Iterating a Map

- To print a map simply use one of the **iterators** inside a **for-of**

```
let iterable = phonebookMap.keys();
for (let key of iterable) {
 console.log(` ${key} => ${phonebookMap.get(key)} `);
}

for (let [key, value] of phonebookMap)
for (let [key, value] of phonebookMap.entries()) {
 console.log(` ${key} => ${value} `);
}
```

Примерна задача решена:

```
function solve(input = []) {
 const carBrands = new Map();
 input.forEach(el => {
 let [brand, model, count] = el.split(" | ");
 count = Number(count);

 if (carBrands.has(brand)) {
 let carBrand = carBrands.get(brand);
 if (carBrand.has(model)) {
 let carModel = carBrand.get(model);
 carBrand.set(model, carModel + count);
 } else {
 carBrand.set(model, count);
 }
 } else {
 const modelMap = new Map();
 modelMap.set(model, count);
 carBrands.set(brand, modelMap);
 }
 });
}

let iterable = Array.from(carBrands.entries());

for (const brandKey of carBrands.keys()) {
 console.log(brandKey);
 const brand = carBrands.get(brandKey);
 for (const [modelKey, count] of brand.entries()) {
 for (const [modelKey, count] of brand) {
 console.log(`### ${modelKey} -> ${count}`);
 }
 }
}
```

```

}

solve(['Audi | Q7 | 1000',
'Audi | Q6 | 100',
'BMW | X5 | 1000',
'BMW | X6 | 100',
'Citroen | C4 | 123',
'Volga | GAZ-24 | 1000000',
'Lada | Niva | 1000000',
'Lada | Jigula | 1000000',
'Citroen | C4 | 22',
'Citroen | C5 | 10']
);

```

## Map Sorting

- To **sort** a Map, first transform it into an **array**
- Then use the **sort()** method

```

let map = new Map();
map.set("one", 1);
map.set("eight", 8);
map.set("two", 2);
let sorted = Array.from(map.entries())
 .sort((a, b) => a[1] - b[1]); // Sort ascending by value

for (let kvp of sorted) {
 console.log(` ${kvp[0]} -> ${kvp[1]}`);
}

```

## What is a Set?

- Store **unique values** of any type, whether **primitive** values or **object** references
- Set objects are **collections** of values

```

let set = new Set([1, 2, 2, 4, 5]);
// Set(4) { 1, 2, 4, 5 }
set.add(7); // Add value
console.log(set.has(1));
// Expected output: true

```

- Can **iterate** through the elements of a set in **insertion** order

## WeakMap and WeakSet

Java, JS, C#, Python are **automatic memory management systems**

- **Special variants** of Map and Set
- Their elements **do not** count as **active** references - Елементите в тях не се броят за активни референции от garbage-колектора
  - Reference types visible (in **scope**) in the program stack are **active**

- Active references **remain in memory**
- **Out-of-scope** references are **removed by the garbage collector**
- These collections are used in **memory-intensive** applications

## 18.4. Modal

Modal – нещо като pop-up прозорец, който обаче блокира/спира изпълнението на приложението/браузъра

```

<style>
 .demo-modal {
 position: fixed;
 top: 0;
 bottom: 0;
 left: 0;
 right: 0;
 z-index: 100;
 background-color: rgb(80, 80, 80, 0.315);
 }

 .modal-dialog { //клас
 display: block;
 width: 300px;
 margin: auto;
 margin-top: vh;
 background-color: white;
 box-shadow: 5px 5px 15px 5px #00000050;
 padding: 32px;
 text-align: center;
 }

 .modal-dialog>span { //клас modal-dialog с елемент span
 display: block;
 margin-bottom: 32px;
 }
</style>
</head>

<body>
//Това нещо ще го създадем динамично чрез JS
<div class="demo-modal">
 <div class="modal-dialog">
 This is a message
 <button>OK</button>
 </div>
</div>
///////
</body>
```

```

class Modal {
 constructor(message = "Alert") {
 this.message = message;
 this.element = this._init();
 this.render();
 }

 _init() {
 const overlay = createMyElement('div', { className: 'demo-modal' },
 createMyElement('div', { className: 'modal-dialog' },
 createMyElement('span', {}, this.message),
 button('OK', this.onClose.bind(this)))
);
 return overlay;
 }

 render() {
 document.body.appendChild(this.element);
 }

 onClose() {
 this.element.remove();
 }
}

function start() {
 new Modal("This is a message");
 // alert(); - няма стилизация alert-а, и браузърт може да блокира ако викаме много
 // alert-и
}

document.querySelector('button').addEventListener('click', start);

```

Функция за създаване на DOM елементи с поделементи

```

Function e createElement(type, props, ...content) {
 const element = document.createElement(type);

 // слагане на properties/attributes
 for (let prop in props) {
 element[prop] = props[prop];
 }

 // слагане на поделементи на елемента
 for (let entry of content) {
 if (typeof entry == 'string' || typeof entry == 'number') {
 entry = document.createTextNode(entry); // правим го на текст
 }
 element.appendChild(entry);
 }
}

```

```

 }

 return element;
}

function button(label, callback) {
 const element = createMyElement('button', {}, label);
 element.addEventListener('click', callback);
 return element;
}

```

Същата функция за създаване на DOM елементи

```

//function html that creates elements
function e(type, attri, ...content) {
 const element = document.createElement(type);

 for (let prop in attri) {
 element[prop] = attri[prop];
 }

 for (let item of content) {
 if (typeof item == 'string' || typeof item == 'number') {
 item = document.createTextNode(item);
 // item = document.createElement(item);
 }
 element.appendChild(item);
 }

 return element;
}

//function htlm that creates elements - another version -
export function e(type, attributes, ...content) {
 const result = document.createElement(type);

 for (let [attr, value] of Object.entries(attributes || {})) {
 if (attr.substring(0, 2) == 'on') {
 result.addEventListener(attr.substring(2).toLocaleLowerCase(), value);
 } else {
 result[attr] = value;
 }
 }

 content = content.reduce((a, c) => a.concat(Array.isArray(c) ? c : [c]), []);
 //flatmap

 content.forEach(e => {
 if (typeof e == 'string' || typeof e == 'number') {
 const node = document.createTextNode(e);

```

```
 result.appendChild(node);
 } else {
 result.appendChild(e);
 }
});

return result;
}
```

```

const contactBtn = e('button', {}, 'Contact with owner');

const pet = e('li', {},
 e('p', {},
 e('strong', {}, name),
 ' is a ',
 e('strong', {}, age),
 ' year old ',
 e('strong', {}, kind)),
 e('span', {}, `Owner: ${owner}`),
 contactBtn);

```

▼<ul>

▼<li>

▼<p> == \$0

```

Tom
" is a "
0.3
" year old "
cat
</p>
Owner: Jim King
<button>Contact with owner</button>

▶...


```

```
e('input', { placeholder: 'Enter your names' }, [])
```

## 19. Prototypes and Inheritance

Prototypes, Prototype Chain, Class Inheritance

### 19.1. Internal Object Properties

#### Internal Properties

- Every object field has **four** properties:
  - **Enumerable** - can access to all of them using a **for...in** loop - ако е false, то при обхождане в цикъл и при печатане не е достъпна, но ако директно извикаме полето, то тогава е достъпно
    - Enumerable property are returned using **Object.keys** method
  - **Configurable** - can **modify the behavior** of the property
    - You can delete only **configurable** properties
  - **Writable** - can **modify** their **values** and update a property just assigning a new value to it
  - **Value**

```
{
 name: {
 value: 'Peter',
 }
}
```

```
writable: true,
enumerable: true,
configurable: true
},
age: { value: 28, writable: true, enumerable: true, configurable: true }
}

'use strict'

const person = {
 name: "Peter",
 age: 28
}

console.log(Object.getOwnPropertyDescriptors(person));
```

Прототипът на един обект е референция към друг обект

Добавяне на нови свойства и методи към прототипа на даден обект – чрез  
myObj.prototype.newMethod\_newVariable\_Property

#### Вариант 1:

```
console.log(Object.defineProperty(person, 'name', {
 value: 'Jackson',
 writable: true,
 enumerable: false,
 configurable: true
});

console.log(Object.defineProperty(person, 'lastName'), {
 get() {
 return 'Jackson'
 },
 writable: true,
 enumerable: true,
 configurable: false
});
```

#### Вариант 2:

```
'use strict'

const person = {
 name: "Peter",
 age: 28
}
```

```

Object.defineProperty(person, 'lastName', {
 get() {
 return this._lastName || '';
 },
 set(value){
 this._lastName = value;
 },
 enumerable: true,
 configurable: false
});

```

```

console.log(person);
console.log(person.lastName); // ''

person.lastName = 'Johnson';
console.log(person);
console.log(person.lastName); //Johnson

```

### Object's Non-enumerable Properties

- They won't be in for...in iterations
- They won't appear using Object.keys function
- **They are not serialized when using JSON.stringify**

```

let ob = {a:1, b:2};
ob.c = 3;
Object.defineProperty(ob, 'd', { value: 4, enumerable: false }); //задава ново свойство с име
д на обекта об или променя свойство
ob.d; // => 4
for(let key in ob) console.log(ob[key]); //1 2 3
Object.keys(ob); // => ["a", "b", "c"]
ob; // => {a: 1, b: 2, c: 3, d: 4}
ob.d; // => 4

```

### Object's Non-writable Properties

- Once its value is defined, it is **not possible to change** it using assignments

```

let ob = { a: 1 };
Object.defineProperty(ob, 'B', { value: 2, writable: false });
ob.B; // => 2
ob.B = 10;
ob.B; // => 2

```

- If the non-writable property **contains** an **object**, the **reference** to the object is what is **not writable**, but the **object itself can be modified**

### Object's Non-configurable Properties

- Once you have defined the property as **non-configurable**, there is only **one behavior** you **can change**
  - If the property is **writable**, you can convert it to non-writable

- Any other try of definition update will fail throwing a TypeError

```
let ob = {};
Object.defineProperty(ob, 'a', { configurable: false, writable: true });
Object.defineProperty(ob, 'a', { enumerable: true }); // throws a TypeError
Object.defineProperty(ob, 'a', { value: 12 }); // throws a TypeError
Object.defineProperty(ob, 'a', { writable: false }); // This is allowed!!
Object.defineProperty(ob, 'a', { writable: true }); // throws a TypeError
delete ob.a; // => false
```

## Object Freeze and Seal

```
Object.freeze();
Object.seal();
Object.preventExtensions();
```

The `Object.freeze()` method **freezes** an object. A frozen object can no longer be changed; freezing an object prevents new properties from being added to it, existing properties from being removed, prevents changing the enumerability, configurability, or writability of existing properties, **and prevents the values of existing properties from being changed**. In addition, freezing an object also prevents its prototype from being changed. `freeze()` returns the same object that was passed in.

```
let cat = { name: 'Tom', age: 5 };
Object.freeze(cat);
cat.age = 10; // Error in strict mode
cat.gender = 'male'; // Error in strict mode
console.log(cat); // { name: 'Tom', age: 5 }
```

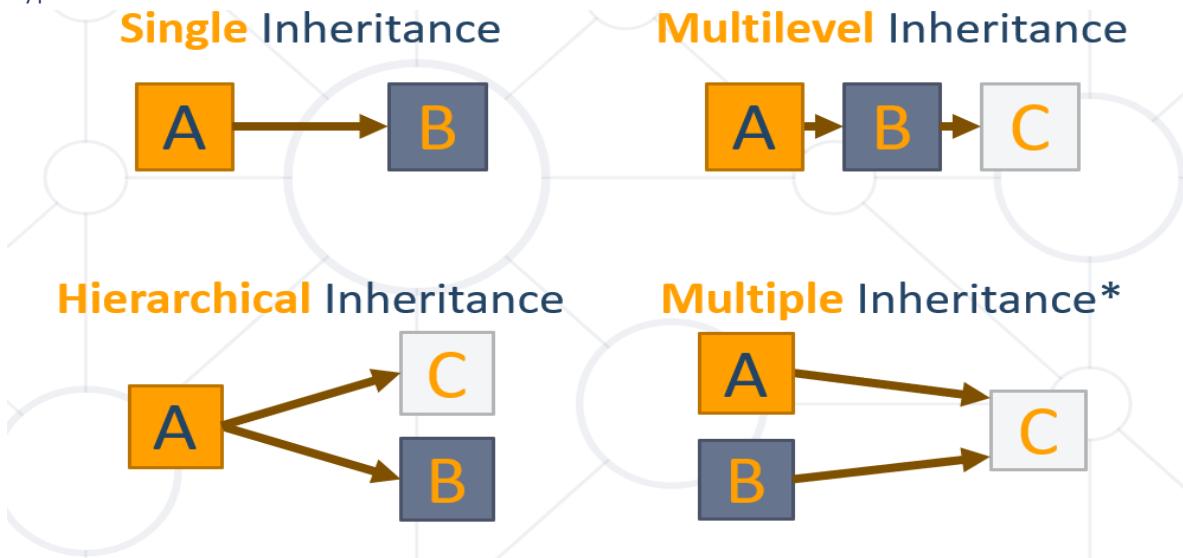
The `Object.seal()` method seals an object, preventing new properties from being added to it and marking all existing properties as non-configurable. Values of present properties can still be changed as long as they are writable.

```
cat = { name: 'Tom', age: 5 };
Object.seal(cat);
cat.age = 10; // OK
delete cat.age; // Error in strict mode
console.log(cat); // { name: 'Tom', age: 10 }
```

## 19.2. Inheritance

Наследяването в JS е само синтактична захар

## Types of Inheritance



\* Not supported in JS with **classes**, but works with **composition**

В JAVA и C# multiple inheritance не е възможно да постигнем! Може да използваме интерфейси, и да ги реализираме/имплементираме, но това е различно от директно наследяване.

Deadly diamond = multiple inheritance with classes in JS

## Класово наследяване в JS

## Прототипно наследяване в JS

### 19.3. The Prototype - Object Delegation

What is a Prototype?

- Every object in JS has a **prototype** (template)
  - Internally called `__proto__` in browsers and NodeJS
  - Properties **lookup** follows the **prototype chain** – ако дадения обект няма дадено свойство, то JS гледа в прототипа на дадения обект, като на върха е прототипа Object, а прототипа на Object е null.
- Obtained with `Object.getPrototypeOf(obj)` – използваме тази функция
- Reference to another objects
  - Objects are **not** separate and disconnected, but **linked** – жива връзка. Ако променим родителя, това ще се отрази и на наследниците

Note: `__proto__` is for debugging and should **never** be used in production code!

- Objects **inherit properties** and **methods** from a **prototype**
- The **prototype property** allows you to add **new properties** to object **constructors**

Краят на веригата от прототипи – за литерал обект

```
const myObj = {};
Object.getPrototypeOf(Object.getPrototypeOf(myObj)); //null
```

Прототипът идва от функцията, която е създала обекта

```
//constructor function - конструктор функция
function Person(first, last, age) {
 this.firstName = first;
 this.lastName = last;
 this.age = age;
}
Person.prototype.nationality = "Bulgarian"; //задаваме на прототипа поле nationality
```

*Static Статично задаване на ново пропърти*

```
Person.middleName = "Petrov"; //за всички инстанции на Person ще има пропърти middleName
```

//Клас с конструктор

```
class Person {
 constructor(first, last, age) {
 this.firstName = first;
 this.lastName = last;
 this.age = age;
 }
}
Person.prototype.nationality = "Bulgarian";
```

Simulated Class Functionality

- Before ES6, **classes** were composed **manually – with constructor function**

```
function Rectangle(width, height) {
 this.width = width;
 this.height = height;
}

Rectangle.prototype.area = function () { //задава на бащата общата функция area
 return this.width * this.height;
}
let rect = new Rectangle(3, 5);
```

По новия начин на създаване на клас – като направим клас JS, реално при интерпретатора на JS се създават конструктор функции като едно време. Затова казваме, че наследяването в JS е само синтактична захар.

```
class Rectangle {
 constructor(width, height) {
 this.width = width;
 this.height = height;
 }
 area() {
 return this.width * this.height;
 }
}
```

## Object Creation

- **Literal** creation
- **Constructor** creation
  - Have an **implicit reference** (prototype) to the value of their constructor's "prototype" property
  - Gets an internal **\_\_proto\_\_** link to the object

### JavaScript Objects

- **Literals – пишем си го директно – без клас и без constructor function**

```
let bar = {
 me: "I am b1",
 speak: function () {
 console.log("Hello, " + this.me + ".");
 }
};
```

- **Constructed – ползваме или клас или constructor function**

```
function Bar(name) {
 this.me = "I am " + name;
 this.speak = function () {
 console.log("Hello, " + this.me + ".");
 }
};

let b1 = new Bar("b1");
```

Данните се пазят в обекта, а функционалността/методите се пазят/са делегирани в прототипа

```
class Circle {
 constructor(radius) {
 this.radius = radius;
 }

 getArea(){
 return this.radius ** 2 * Math.PI;
 }
}

const c = new Circle(5);
console.log(c.hasOwnProperty("getArea")); //false
console.log(c.hasOwnProperty("radius")); //true
```

```
const c = new Circle(5);
console.log(Object.getPrototypeOf(c));
▼ {constructor: f, getArea: f} ⓘ
 ▼ constructor: class Circle
 arguments: (...)

 caller: (...)

 length: 1

 name: "Circle"

 ▼ prototype:
 ► constructor: class Circle
 ► getArea: f getArea()
 ► [[Prototype]]: Object
 [[FunctionLocation]]: VM367:2
 ► [[Prototype]]: f ()
 ► [[Scopes]]: Scopes[2]
 ► getArea: f getArea()
 ► [[Prototype]]: Object

console.log(Object.getPrototypeOf(c).hasOwnProperty("getArea")); //true
```

#### Два едни и същи записи проверяват едно и също нещо

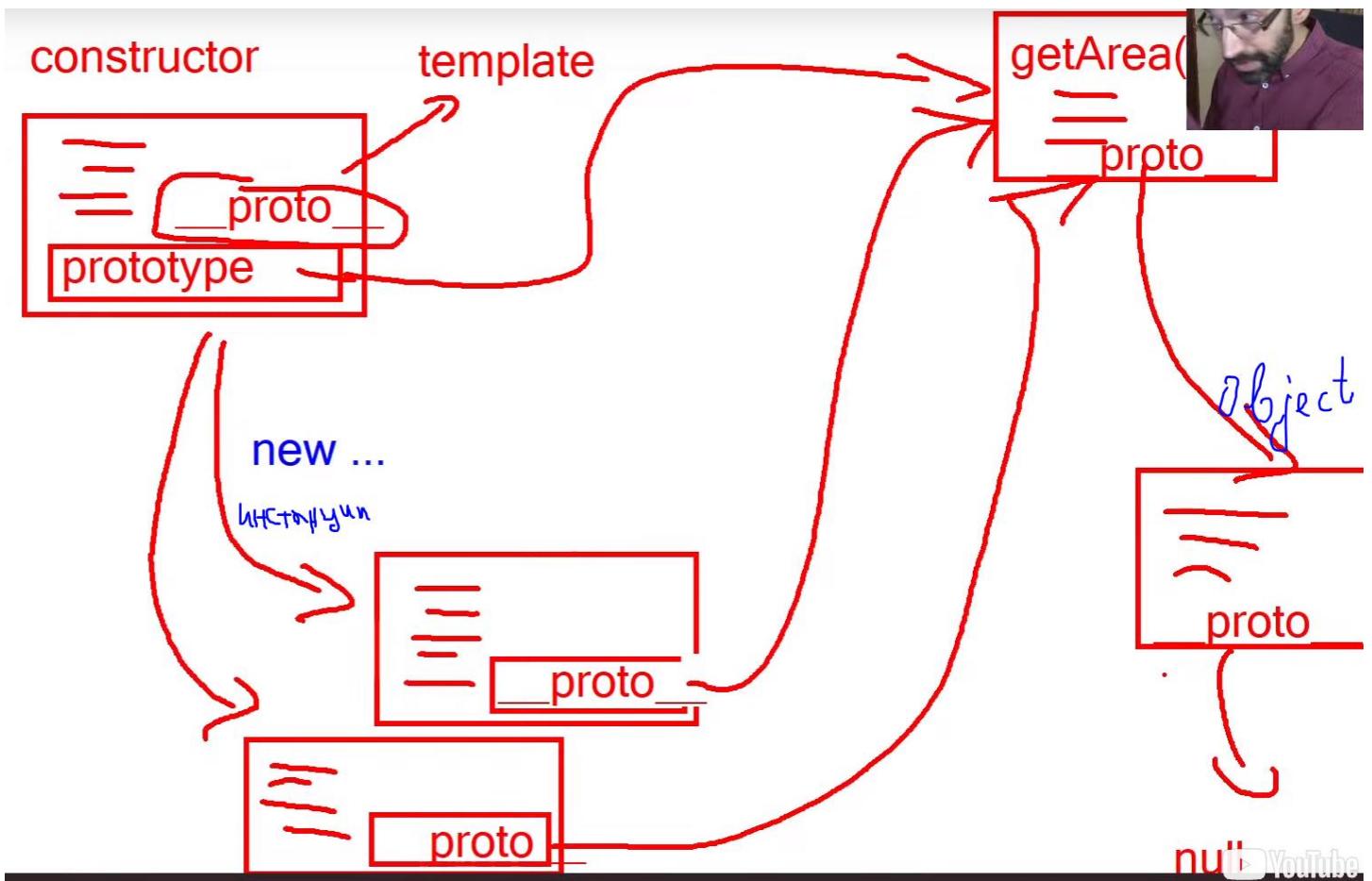
```
//true - дали __proto__ (шаблонът на инстанцията) е прототипът на функцията/на класа
console.log(Object.getPrototypeOf(c) == Circle.prototype);
console.log(c.__proto__ == Circle.prototype);
console.log(c instanceof Circle); //true
```

прототипа се превръща в шаблон \_\_proto\_\_ на новосъздадените неща.

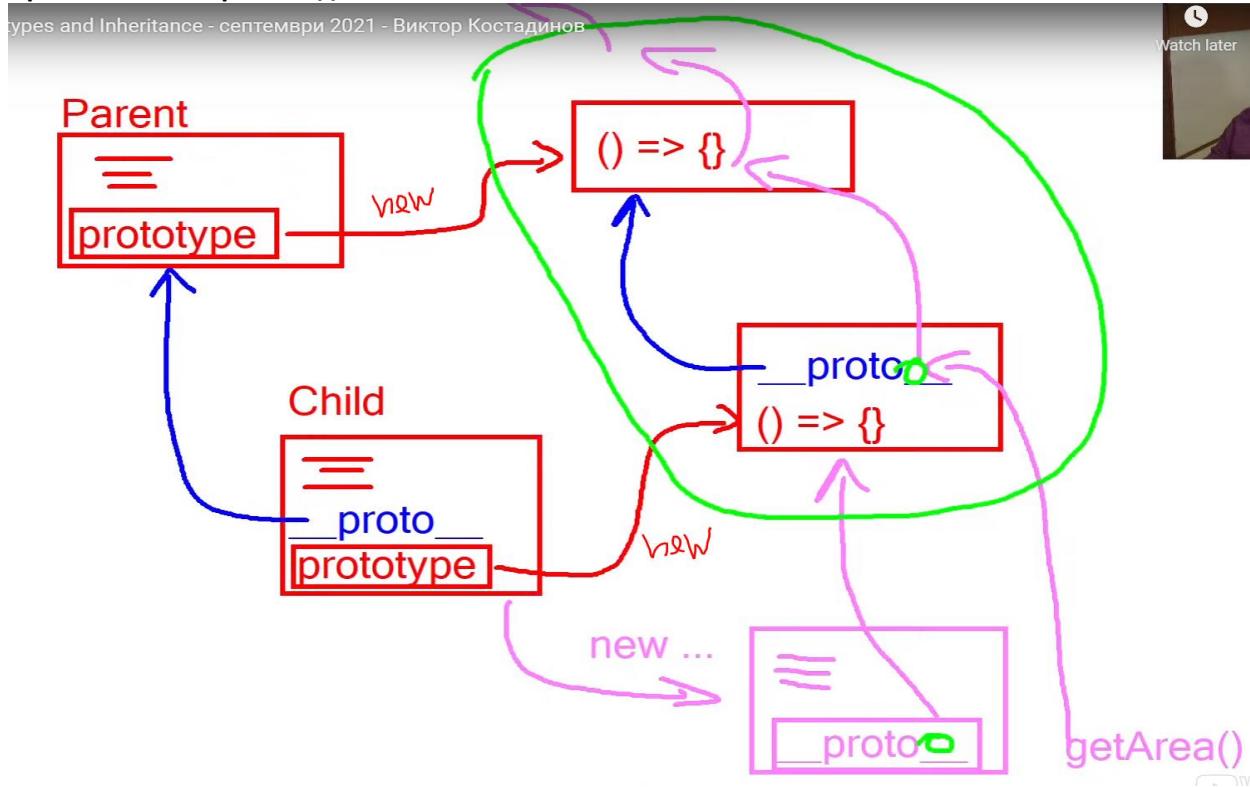
\_\_proto\_\_ на новосъздадените неща се смята за груба грешка ако тръгнем да му променяме свойствата

Като викаме **Object.getPrototypeOf(obj)**, реално викаме \_\_proto\_\_

Един и същи обект може да има както \_\_proto\_\_, така и prototype. И едното е различно от другото.



Прототипната верига се движи по шаблоните обикновено:



Масивът в JS също е обект

```
console.log(Array.isArray([])); //true
console.log([] instanceof Array); //true
> const myArr = []
< undefined
> myArr
< [] ⓘ
 length: 0
 ▼ [[Prototype]]: Array(0)
 ► at: f at()
 ► concat: f concat()
 ► constructor: f Array()
 ► copyWithin: f copyWithin()
 ► entries: f entries()
 ► every: f every()
 ► fill: f fill()
 ► filter: f filter()
 ► find: f find()
 ► findIndex: f findIndex()
 ► flat: f flat()
 ► flatMap: f flatMap()
 ► forEach: f forEach()
 ► includes: f includes()
 ► indexOf: f indexOf()
 ► join: f join()
 ► keys: f keys()
```

Краят на веригата от прототипи – за масив обект

При масива са две нива на наследяване до стигане краят на веригата от прототипи

```
const myArr = [];
console.log(Object.getPrototypeOf(myArr)); //{}
console.log(Object.getPrototypeOf(Object.getPrototypeOf(Object.getPrototypeOf(myArr)))); //null
```

### Object Create

- The **Object.create()** method creates a **new object**, using an existing object as **prototype**

#### Пример 1:

```
const dog = {
 name: 'Sparky',
 printInfo: function () { console.log(`My name is ${this.name}`) }
};

const myDog = Object.create(dog);
myDog.name = 'Max'; // inherited properties can be overwritten
myDog.breed = 'shepherd'; // breed is a property of myDog
myDog.printInfo(); // My name is Max
```

#### Пример 2:

```
const myProto = {
 sayHi(){
 console.log(`${this.name} says hi!`);
 }
}
```

```
const instanc = Object.create(myProto);
instanc.name = "John";
instanc.sayHi(); // John says hi!
```

Можем да си добавяме ново свойство на прототипа, в случая да си направим функция, която да се ползва от фабричния масив. Това не трябва да го правим – да променяме държавни функции.

```
Array.prototype.getLastIndex = function () {
 return this.length - 1;
}
```

Но можем да си създадем наша колекция тип масив, където да заложим дадената функция/свойство

### \_\_proto\_\_ vs Prototype Property

- **\_\_proto\_\_**
  - Property of an objects that **points** at the prototype that has been **set**
  - Using **\_\_proto\_\_** directly is deprecated!
- **prototype**
  - Property of a **function** set if your object is created by a **constructor function**

Objects do not have **prototype** property – свойството **prototype** го има само на/при **constructor function** (при използване на клас реално се вика пак **конструктор функция**)

### Prototype Chain – Simple Example

```
function Foo(y) {
 this.y = y;
}

Foo.prototype.x = 10; //четенето по веригата се предава, но задаването на стойност не се предава по веригата
Foo.prototype.calculate = function (z) {
 return this.x + this.y + z;
};

let b = new Foo(20);
console.log(b.calculate(30)); // 60
```

### Prototype Inheritance = Наследяване на класове по трудния начин

#### Пример 1

```
function Foo(who) {
 this.me = who;
}

Foo.prototype.identify = function () {
 return "I am " + this.me;
}
```

```

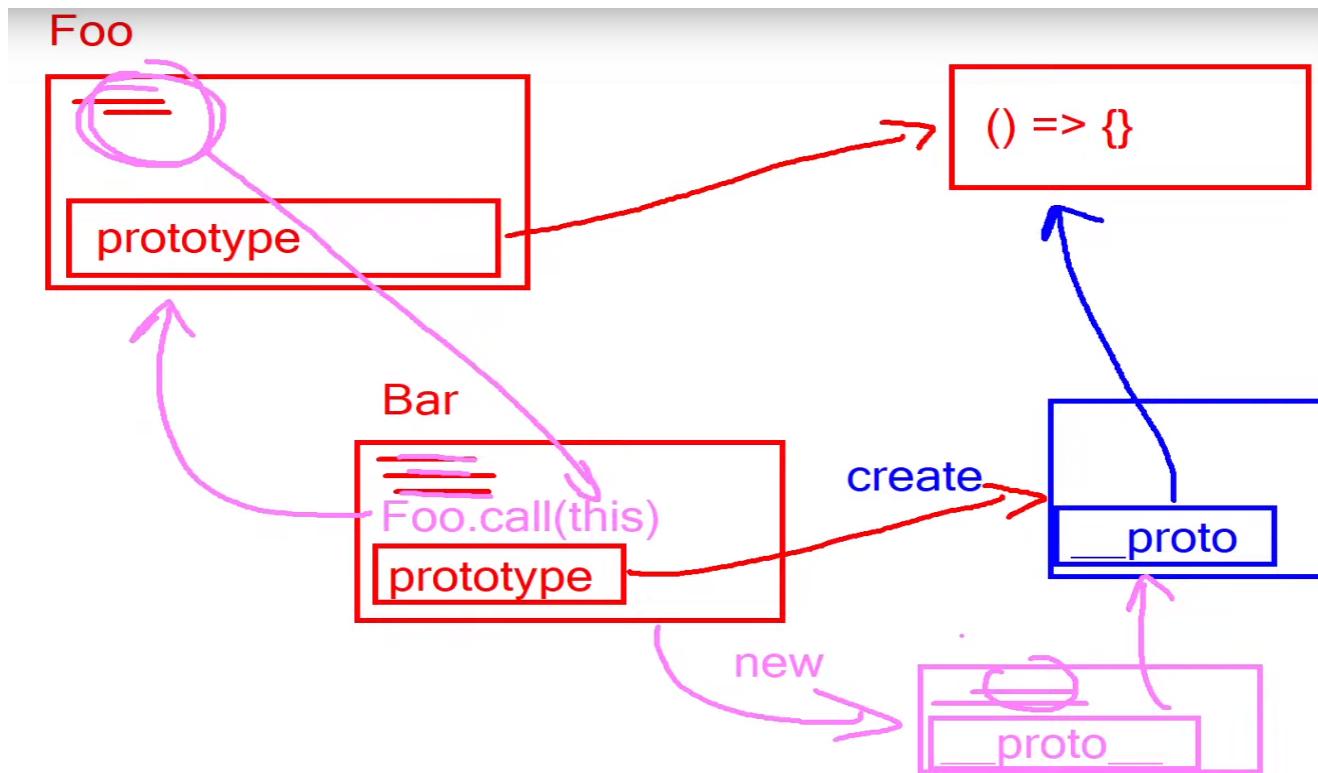
function Bar(who) {
 Foo.call(this, who); //super(who)
}

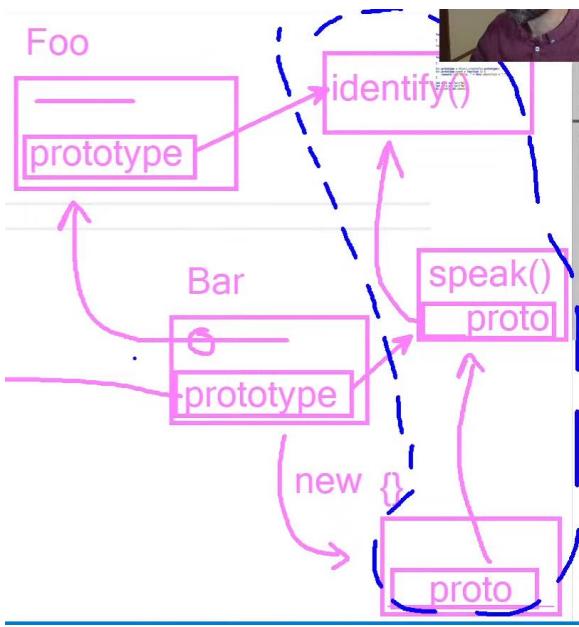
Bar.prototype = Object.create(Foo.prototype); //прототипа на конструктор функцията Bar
наследува прототипа на конструктор функцията Foo

Bar.prototype.speak = function () {
 console.log("Hello, " + this.identify() + ".");
}

let b1 = new Bar("b1");
let b2 = new Bar("b2");
b1.speak();
b2.speak();

```





### Пример 2:

```
function Human(name) {
 this.name = name;
}

Human.prototype.printName = function () {
 console.log(this.name);
}

function Person(name) {
 Human.call(this, name); //super(name)
}
```

Person.prototype = Object.create(Human.prototype); //конструктор функцията Person наследява конструктор функцията Human

```
let human = new Human("Ivan");
human.printName(); //Ivan

let person = new Person("Ivan");
person.printName(); //Ivan
```

Extending prototype of a class

```
class Person {
 constructor(name, email) {
 this.name = name;
 this.email = email;
 }
 toString() {
 let className = this.constructor.name;
 return `${className} (name: ${this.name}, email: ${this.email})`;
 }
}
```

```

 }
}

function extendPrototypeFunct(classDefinition) {
 classDefinition.prototype.species = "Human";
 classDefinition.prototype.toSpeciesString = function () {
 return `I am a ${this.species}. ${this.toString()}`;
 }
 //this.species вика на текущата функция value-то на species
 //this.toString() вика на текущата функция, която вика на класа Person toString() метода
}

extendPrototypeFunct(Person);
let p = new Person("Pesho", "email@hit.bg");
console.log(p.species);
console.log(p.toSpeciesString());

```

Extending prototype of a class when we already have some instances created

Можем да екстендираме прототипа на класовата дефиниция, след като са създадени вече инстанции

```

class Teacher {
 constructor(name, email, subject) {
 this.name = name;
 this.email = email;
 this.subject = subject;
 }

 toString() {
 let className = this.constructor.name;
 return `${className} (name: ${this.name}, email: ${this.email})`;
 }
}

function extendPrototypeFunct(classDefinition) {
 classDefinition.prototype.species = "Human";
 classDefinition.prototype.toSpeciesString = function () {
 return `I am a ${this.species}. ${this.toString()}`;
 }
}

let t = new Teacher("John", "email@hit.bg", "Math");
console.log("before", t.species); //undefined
extendPrototypeFunct(Teacher);
console.log("after", t.species); //Human

```

Extensible Object

```

function extensibleObject() {
 return {

```

```

 extend: function(template) {
 let objProto = Object.getPrototypeOf(this);
 let templateEntries = Object.entries(template);
 for(const [key, value] of templateEntries){
 if (typeof value == "function") {
 objProto[key] = value; //към прототипа на текущия обект
 } else {
 this[key] = value; //към текущия обект
 }
 }
 }
 }

const template = {
 extensionMethod: function () { },
 extensionProperty: 'someString'
}

const myObj = extensibleObject();
myObj.extend(template);
console.log(myObj);

```

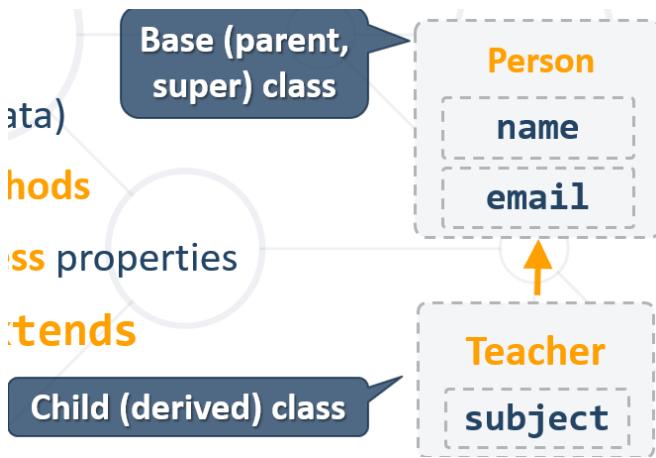
## 19.4. Class Inheritance (ES6 - ECMAScript 6)

### Traditional Classes

- Classes are a **design pattern**
- Classes mean - creating **copies**
  - When **instantiated** – a **copy** from class to instance
  - When **inherited** – a **copy** from parent to child
- Class inheritance is a powerful tool, but has many **drawbacks** and **limitation**
  - **Composition** should be **preferred** whenever possible!

### Class Inheritance

- Classes can **inherit** (extend) other classes
  - **Child class** inherits **data + methods** from its parent
- **Child class** can:
  - Add **properties** (data)
  - Add / replace **methods**
  - Add / replace **access** properties
- Use the keyword **extends**
- **Като променим прототипа на базовия клас, всички наследници ще бъдат с променени /обновени прототипи. Да се избягва ни съветват – голяма сила и голяма отговорност.**
- **Не можем да наследим от два класа едновременно**



```

class Person {
 constructor(name, email) {
 this.name = name;
 this.email = email;
 }
 toString() {
 let className = this.constructor.name;
 return `${className} (name: ${this.name}, email: ${this.email})`;
 }
}

class Teacher extends Person {
 constructor(name, email, subject){
 super(name, email);
 //Person.call(this, name, email); - като super работи
 this.subject = subject;
 }

 //overriding the toString() method
 toString() {
 return this.subject + " " + super.toString();
 }
}

let t = new Teacher("John", "email@hit.bg", "Math");
console.log(t); // Teacher { name: 'John', email: 'email@hit.bg', subject: 'Math' }
console.log(t.toString()); // Math Teacher (name: John, email: email@hit.bg)
console.log(t instanceof Teacher); //true
console.log(t instanceof Person); //true - в JS инстанцията е и инстанция на бащиния клас
също

```

## Classes in JavaScript

- **Prototypal inheritance** instead of classical inheritance
- **Does not automatically** create copies
- Common keys and values are shared by **reference**

- Delegates not blueprints! – делегиране, а не шаблонизиране

Overriding to String()

*Option 1 – create complete new toString exit data*

```

class Person {
 constructor(name, email) {
 this.name = name;
 this.email = email;
 }
 toString() {
 let className = this.constructor.name;
 return `${className} (name: ${this.name}, email: ${this.email})`;
 }
}

class Teacher extends Person {
 constructor(name, email, subject) {
 super(name, email);
 // Person.call(this, name, email);
 this.subject = subject;
 }

 //overriding the toString() method
 toString() {
 let className = this.constructor.name;
 return `${className} (name: ${this.name}, email: ${this.email}, subject:
${this.subject})`;
 }
}

class Student extends Person {
 constructor(name, email, course) {
 super(name, email);
 this.course = course;
 }

 //overriding the toString() method
 toString() {
 let className = this.constructor.name;
 return `${className} (name: ${this.name}, email: ${this.email}, course:
${this.course})`;
 }
}

let p = new Person("John", "email@hit.bg");
console.log(p.toString());

let t = new Teacher("John", "email@hit.bg", "Math");
console.log(t.toString());

```

```
let s = new Student("John", "email@hit.bg", 7);
console.log(s.toString());
```

Option 2 - reuse code by using the **toString()** function of the base class

```
class Person {
 constructor(name, email) {
 this.name = name;
 this.email = email;
 }
 toString() {
 let className = this.constructor.name;
 return `${className} (name: ${this.name}, email: ${this.email})`;
 }
}

class Teacher extends Person {
 constructor(name, email, subject) {
 super(name, email);
 // Person.call(this, name, email);
 this.subject = subject;
 }

 //overriding the toString() method
 toString() {
 let baseClassString = super.toString(); //тук инстанцията извиква името на класа,
 от която е инстанцирана!!!!
 let zzz = baseClassString.substring(0, baseClassString.length-1);
 return zzz + `, subject: ${this.subject})`;
 }
}

let p = new Person("John", "email@hit.bg");
console.log(p.toString());

let t = new Teacher("John", "email@hit.bg", "Math");
console.log(t.toString());
```

Creating an abstract class

В JS няма абстрактен клас предефиниран!!!

Пример 1

```
function Hierarchy() {
 class Figure {
 constructor(units = 'cm') {
 this.units = units;
 if (new.target === Figure) { //тук има врътка
```

```
 throw new TypeError('Figure class is abstract');
 }
}
}
```

## Пример 2

```
class Employee {
 constructor(name, age) {
 if (new.target === Employee) { //тук има врътка
 return new Error('Employee class must be abstract');
 }
 this.name = name;
 this.age = age;
 }
}
```

## 20. Workshop new info

## 20.1. Инсталиране на Servers

## Lite server

## Инсталираме сървър

```
npm install -g lite-server
```

**Пускаме сървъра от текущата папка:**

## lite-server

[Browsersync] Access URLs:

Local: <http://localhost:3000>  
External: <http://192.168.100.4:3000>

```
UI: http://localhost:3001
UI External: http://localhost:3001

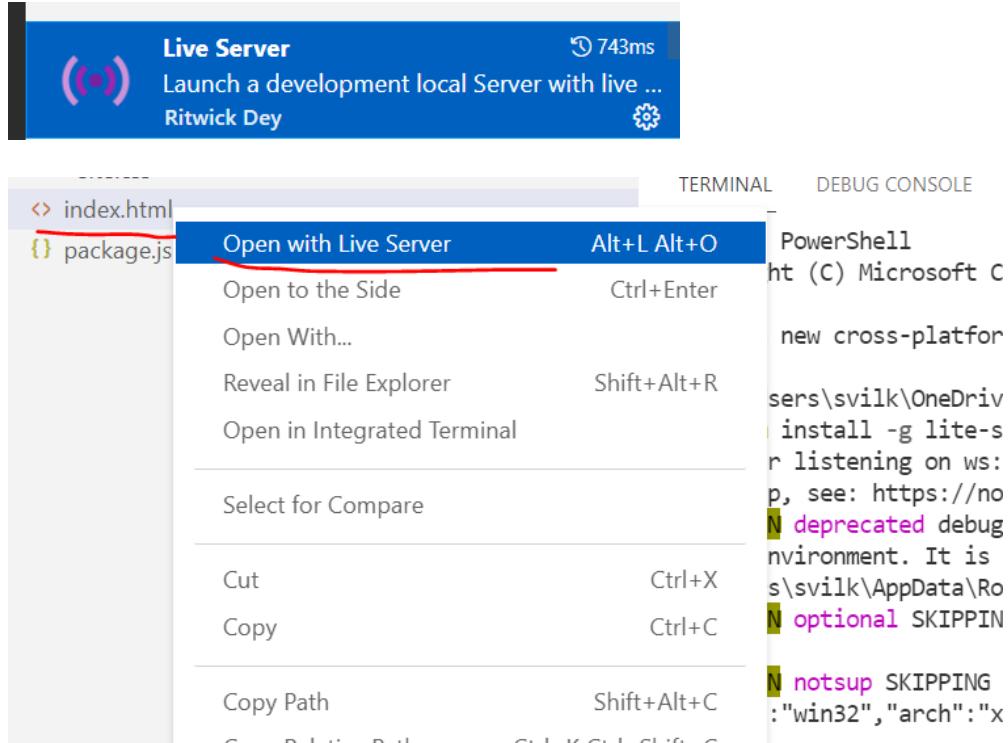
[Browsersync] Serving files from: './'
[Browsersync] Watching files...
21.11.08 22:55:41 200 GET /index.html
21.11.08 22:55:41 200 GET /static/site.css
21.11.08 22:55:41 200 GET /src/app.js
21.11.08 22:55:42 200 GET /src/board.js
21.11.08 22:55:42 200 GET /src/import.js
21.11.08 22:55:42 200 GET /src/timer.js
21.11.08 22:55:42 404 GET /favicon.ico
21.11.08 23:00:43 304 GET /index.html
21.11.08 23:00:43 304 GET /static/site.css
21.11.08 23:00:43 304 GET /src/app.js
21.11.08 23:00:43 304 GET /src/board.js
21.11.08 23:00:43 304 GET /src/import.js
21.11.08 23:00:43 304 GET /src/timer.js
```

## Стартираме от Browser-а

<http://localhost:3000/>

Localhost инструкира браузъра да изпраща заявка към същия компютър

## Live Server



<http://127.0.0.1:5500/index.html>

Автоматично каквото save-нем, се опреснява и на html страницата

## Http server

`package.json`

```
{
 "name": "movies",
 "version": "1.0.0",
 "description": "",
 "main": "index.js",
 "scripts": {
 "start": "http-server -a localhost -p 3000 -P http://localhost:3000? -c-1",
 "server": "cd server & node server.js"
 },
 "keywords": [],
 "author": "",
 "license": "ISC",
 "dependencies": {
 "http-server": "^14.1.0",
 "lit-html": "^2.2.3",
 "page": "^1.11.6"
 }
}
```

```
npm i -y
npm i http-server lit-html page
npm run server за backend-a (server и база данни все едно)
npm run start или само npm start за тестване на front-end-a
```

## 20.2. Използване на module import / export

```
<script type="module" src="/src/app.js"></script>
```

JS app.js

```
// import { p1 } from './puzzles.js';
import { generateBoard, button } from './board.js';
import { init } from './import.js';
import { createTimer } from './timer.js';
```

JS board.js X

```
export function generateBoard(values, main) {

}
```

```
export function e(type, attr, ...content) {
```

```
}
```

```
export function button(label, callback) {
```

```
}
```

## 20.3. RequestAnimationFrame() - може да забавя рендерирането на экрана

The `window.requestAnimationFrame()` method tells the browser that you wish to perform an animation and requests that the browser calls a specified function to update an animation before the next repaint. The method takes a callback as an argument to be invoked before the repaint.

You should call this method whenever you're ready to update your animation onscreen. This will request that your animation function be called before the browser performs the next repaint. The number of callbacks is usually 60 times per second, but will generally match the display refresh rate in most web browsers as per W3C recommendation. `requestAnimationFrame()` calls are paused in most browsers when running in background tabs or hidden `<iframe>`s in order to improve performance and battery life.

The callback method is passed a single argument, a `DOMHighResTimeStamp`, which indicates the current time (based on the number of milliseconds since `time origin`).

Пример 1:

```
export function createTimer() {
 const output = document.getElementById('timer');
 let lastTime = performance.now(); - работи само в Browser-а
 let elapsed = 0;
```

```

let active = true;

tick(lastTime);

return {
 pause,
 resume
}

function pause() {
 active = false;
}

function resume() {
 active = true;
 lastTime = performance.now();
 tick(lastTime);
}

function tick(time) {
 const delta = time - lastTime;
 elapsed += delta;
 lastTime = time;

 const total = elapsed / 1000;
 const seconds = total % 60;
 const minutes = (total / 60) % 60;
 const hours = total / 3600;

 output.textContent =
` ${formatTime(hours)}:${formatTime(minutes)}:${formatTime(seconds)} `;
 if (active) {
 requestAnimationFrame(tick);
 }
}

function formatTime(value) {
 return ('0' + Math.floor(value)).slice(-2);
}

```

Пример 2:

```

const element = document.getElementById('some-element-you-want-to-animate');
let start, previousTimeStamp;

function step(timestamp) {
 if (start === undefined)
 start = timestamp;
 const elapsed = timestamp - start;

```

```

if (previousTimeStamp !== timestamp) {
 // Math.min() is used here to make sure the element stops at exactly 200px
 const count = Math.min(0.1 * elapsed, 200);
 element.style.transform = 'translateX(' + count + 'px)';
}

if (elapsed < 2000) { // Stop the animation after 2 seconds
 previousTimeStamp = timestamp
 window.requestAnimationFrame(step);
}
}

window.requestAnimationFrame(step);

```

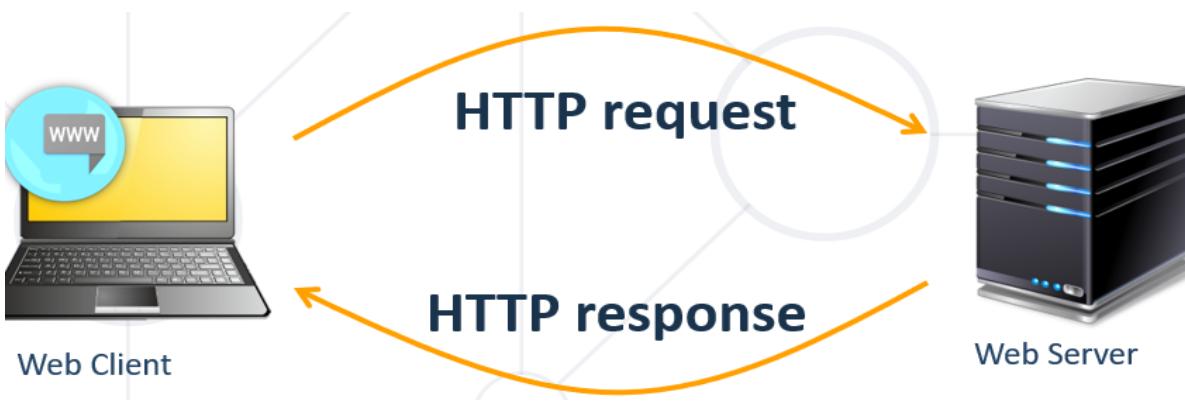
## JS Applications

## 21. HTTP and REST Services

### 21.1. HTTP Protocol Overview

#### HTTP Basics

- **HTTP (Hyper Text Transfer Protocol)**
  - Text-based client-server protocol for the Internet
  - For transferring Web resources (HTML files, images, styles, etc.)
  - Request-response based



#### HTTP Request Methods

- **HTTP** defines **methods** to indicate the desired action to be performed on the identified resource

#### Method on the identified resource

Method	Description
GET	Retrieve / load a resource
POST	Create / store a resource

HTML file  
Image  
Stylesheet  
Media .

Method	Description
GET	Retrieve / load a resource
POST	Create / store a resource
PUT	Update a resource
DELETE	Delete (remove) a resource
PATCH	Update resource partially
HEAD	Retrieve the resource's headers
OPTIONS	Returns the HTTP methods that the server supports for the specified URL

HTTP GET Request - example

```

GET /users/testnakov/repos HTTP/1.1
Host: api.github.com
Accept: */*
Accept-Language: en
Accept-Encoding: gzip, deflate
User-Agent: Mozilla/5.0 (Windows NT 10.0; WOW64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/54.0.2840.71 Safari/537.36
Connection: Keep-Alive
Cache-Control: no-cache
<CRLF>

```

**HTTP request line**

**HTTP headers**

**The request body is empty**

### HTTP POST Request – Example

POST /repos/testnakov/test-nakov-repo/issues HTTP/1.1  
Host: api.github.com  
Accept: \*/\*  
Accept-Language: en  
Accept-Encoding: gzip, deflate  
User-Agent: Mozilla/4.0 (compatible;MSIE 6.0; Windows NT 5.0)  
Connection: Keep-Alive  
Cache-Control: no-cache  
<CRLF>  
{"title": "Found a bug",  
 "body": "I'm having a problem with this.",  
 "labels": ["bug", "minor"]}  
<CRLF>

The request body holds the submitted data

HTTP headers

HTTP request line

POST http://localhost:3030/jsonstore/phonebook

Params Authorization Headers (8) Body Pre-request Script Tests Settings

none  form-data  x-www-form-urlencoded  raw  binary  GraphQL **JSON**

```
1 {
2 "name": "Maya Brezni",
3 "phone": "123456"
4 }
```

*HTTP Response – Example*

HTTP/1.1 200 OK

HTTP response status line

Date: Fri, 11 Nov 2016 16:09:18 GMT+2

Server: Apache/2.2.14 (Linux)

Accept-Ranges: bytes

Content-Length: 84

Content-Type: text/html

HTTP response  
headers

<CRLF>

<html>

<head><title>Test</title></head>

<body>Test HTML page.</body>

</html>

HTTP response body

*HTTP Patch - Example*

PATCH  http://localhost:3030/jsonstore/phonebook/2d5ae478-87c7-45fa-acf9-f04aa4724421

Params Authorization Headers (8) **Body**  Pre-request Script Tests Settings

none  form-data  x-www-form-urlencoded  raw  binary  GraphQL **JSON**

```
1 {
2 ... "phone": "new number 12345"
3 }
```

Body Cookies Headers (6) Test Results 🌐 Status: 200 OK Ti

Pretty Raw Preview Visualize **JSON**

```
2 "2d5ae478-87c7-45fa-acf9-f04aa4724421": {
3 "person": "Maya",
4 "phone": "+1-555-7653",
5 "_id": "2d5ae478-87c7-45fa-acf9-f04aa4724421"
6 },
```

```
1 {
2 "person": "Maya",
3 "phone": "new number 12345",
4 "_id": "2d5ae478-87c7-45fa-acf9-f04aa4724421"
5 }
```

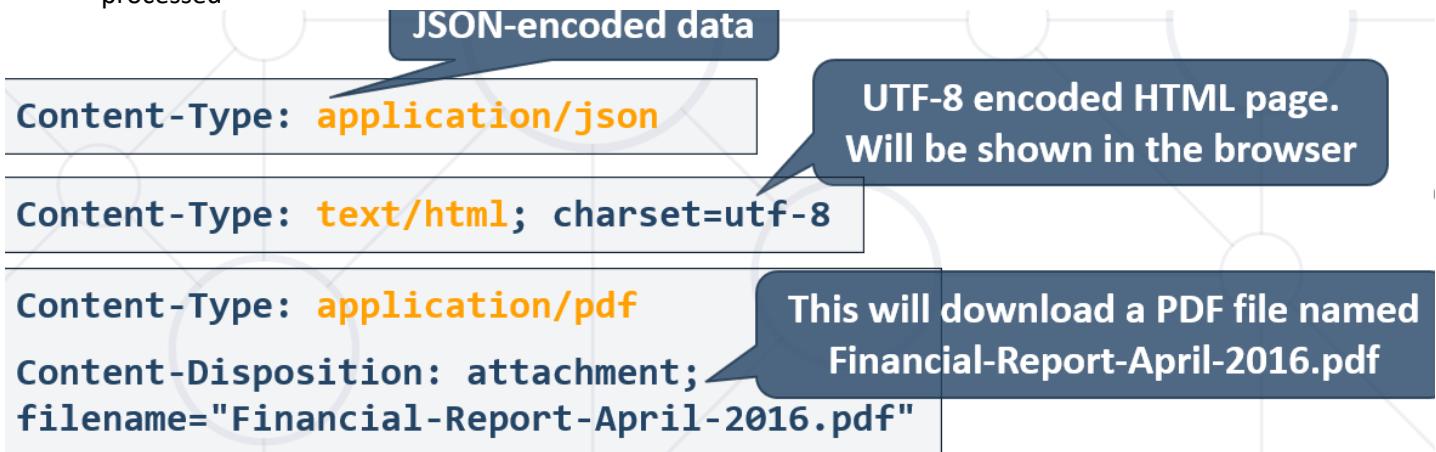
HTTP Response Status Codes

<https://http.cat/>

Status Code	Action	Description
200	OK	Successfully retrieved resource
201	Created	A new resource was created
204	No Content	Request has nothing to return
301 / 302	Moved	Moved to another location (redirect)
400	Bad Request	Invalid request / syntax error
401 / 403	Unauthorized	Authentication failed / Access denied
404	Not Found	Invalid resource
409	Conflict	Conflict was detected, e.g. duplicated email
500 / 503	Server Error	Internal server error / Service unavailable

#### Content-Type and Disposition

- The **Content-Type** / **Content-Disposition** headers specify how the HTTP request / response body should be processed



## 21.2. HTTP Developer Tools

Заявките в браузъра – в един браузър има по много заявки наведнъж

The screenshot shows the Network tab of the Google Chrome DevTools. At the top, there are tabs for Bookmarks, Dict, Online platforms, SoftUni, LCW, and other SOFT Acad. Below the tabs, there are icons for Stop, Refresh, Elements, Console, Sources, Network (which is underlined in red), and a message center with 2 notifications. There are also settings for Preserve log, Disable cache (which is checked and highlighted with a red arrow), and No throttling.

The main area displays a waterfall chart and a table of network requests. The waterfall chart shows several requests as vertical bars. The table has columns: Name, Method, Status, Type, Initiator, Size, Time, and Waterfall (highlighted with a red arrow). The requests listed include various scripts and images from sources like gtm.js, VM86, and styles?.

Name	Method	Status	Type	Initiator	Size	Time	Waterfall
9T19204022...	GET	200	script	gtm.js...	0...	1...	
collect?t=dc...	POST	200	xhr	analyti...	2...	1...	
js?id=G-ZJH...	GET	200	script	gtm.js...	6...	1...	
insight.min.js	GET	200	script	gtm.js...	2....	4...	
tag.js	GET	200	script	VM86...	6...	3...	
4jwrpsxol	GET	200	script	VM86...	7...	1...	
fb-icon.svg	GET	200	svg+...	styles?...	8...	5...	
twitter-icon....	GET	200	svg+...	styles?...	1....	5...	
youtube-ico...	GET	200	svg+...	styles?...	9...	4...	
github-icon....	GET	200	svg+...	styles?...	3....	5...	

At the bottom, a summary bar shows: 89 requests | 1.6 MB transferred | 3.7 MB resources | Finish: 1.2 min | DOMContentLoaded. A red circle highlights the '89 requests' part. A red arrow points to the 'Waterfall' column header. Another red arrow points to the 'Disable cache' checkbox in the top right.

The screenshot shows the Chrome DevTools Network tab with a red arrow pointing to the tab header. Below, a specific request is selected, and its details are shown in the "General" section of the "Request" panel. The request URL is `https://softuni.bg/Content/fonts/redesign/Lato-Heavy.woff2`. The status code is 200 OK (from memory cache). The remote address is 217.174.159.34:443. The referrer policy is strict-origin-when-cross-origin.

Name  
70548619?wmode=0&vv-  
collect  
70548619?wmode=0&vv-  
collect  
70548619?wmode=0&vv-  
105 requests | 258 kB transfer  
Console Issues What's New

Request URL: `https://softuni.bg/Content/fonts/redesign/Lato-Heavy.woff2`  
Request Method: GET  
Status Code: 200 OK (from memory cache)  
Remote Address: 217.174.159.34:443  
Referrer Policy: strict-origin-when-cross-origin

Include third-party cookie issues | X 0 !

## Postman REST Client

The screenshot shows the Postman REST Client interface. On the left, there is a sidebar with collections like Firebase and GitHub. In the center, a request is being built for "List User Repos". The method is GET, the URL is `https://api.github.com/users/testnakov/repos`, and the response status is 200 OK with a time of 165 ms. The response body is displayed in JSON format, showing a list of repositories for the user testnakov.

Filter Collections History All Me Team

GET List User Repos

GET Get Commits by Repo  
GET Get Issues by Repo  
GET Get Issue #1 by Repo  
GET Get Issue #1 Labels by Repo  
POST Create an Issue by Repo  
PATCH Edit Issue #6 by Repo  
PATCH Close Issue #6 by Repo

Firebase 12 requests  
GitHub 8 requests  
Kinvey 8 requests  
Postman Echo 21 requests

Builder Team Library OFFLINE Sign In

Create an Issue b... Get Issue #1 by R... Edit Issue #6 by R... List User Repos No Environment

Send Save Code

Type No Auth

Body Cookies Headers (22) Tests Status: 200 OK Time: 165 ms

Pretty Raw Preview JSON

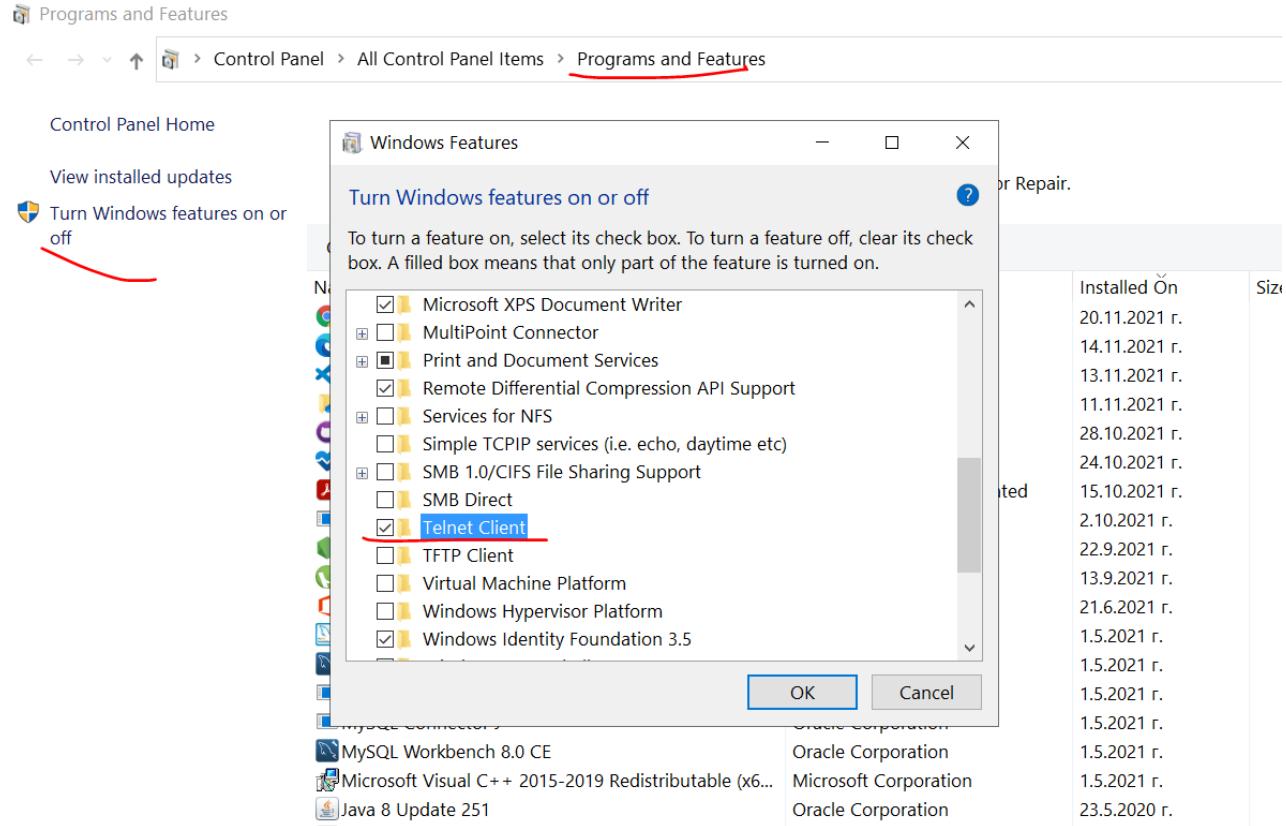
```
[{"id": 73502401, "name": "Flappy-Nakov", "full_name": "testnakov/Flappy-Nakov", "owner": {"login": "testnakov", "id": 23406465, "avatar_url": "https://avatars.githubusercontent.com/u/23406465?v=3", "gravatar_id": ""}, "permissions": {"admin": false, "push": false, "pull": true}, "created_at": "2015-01-12T10:10:41Z", "updated_at": "2015-01-12T10:10:41Z", "clone_url": "https://github.com/testnakov/Flappy-Nakov.git", "ssh_url": "git@github.com:testnakov/Flappy-Nakov.git", "html_url": "https://github.com/testnakov/Flappy-Nakov", "url": "https://api.github.com/repos/testnakov/Flappy-Nakov"}]
```

[Read more about Postman REST Client](#)

## Insomnia REST Client

<https://insomnia.rest/>

## Telnet as a Rest Client – тегаво се работи с Telnet



```
Telnet
Welcome to Microsoft Telnet Client
GET
Escape Character is 'CTRL+['
Microsoft Telnet>
Microsoft Telnet> o www.softuni.bg
Connecting To www.softuni.bg...Could not open connection to the host
Microsoft Telnet> o www.softuni.bg 80
Connecting To www.softuni.bg...
Connection to host lost.

Microsoft Telnet> www.softuni.bg 80
Invalid Command. type ?/help for help
Microsoft Telnet> o https://www.softuni.bg 80
Connecting To https://www.softuni.bg...Could not open connection to the host
Microsoft Telnet> o www.softuni.bg 80
Connecting To www.softuni.bg...
```

```
HTTP/1.1 301 Moved Permanently
Server: nginx/1.18.0 (Ubuntu)
Date: Mon, 25 Oct 2021 15:59:02 GMT
Content-Type: text/html
Content-Length: 178
Connection: keep-alive
Location: https://softuni.bg/

<html>
<head><title>301 Moved Permanently</title></head>
<body>
<center><h1>301 Moved Permanently</h1></center>
<hr><center>nginx/1.18.0 (Ubuntu)</center>
</body>
</html>

t
Connection to host lost.

Press any key to continue...
```

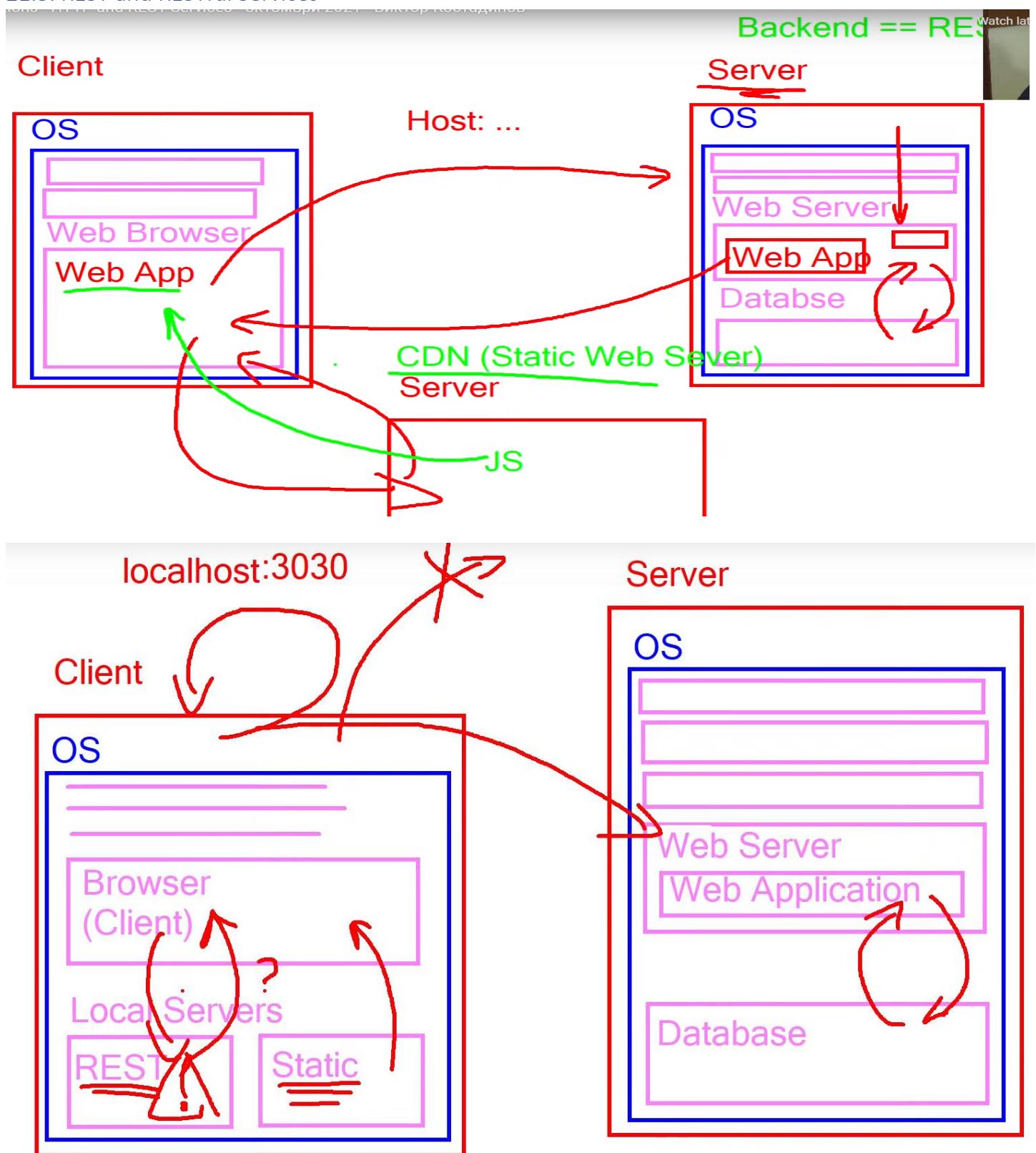
През Ubuntu (Linux) и SSL връзка - as a Rest Client

443 е за криптирана връзка

```
viktorpts@AUGUSTUS:~$ openssl
OpenSSL> s_client -connect softuni.bg:443
```

```
GET / HTTP/1.1
Host: softuni.bg
```

## 21.3. REST and RESTful Services



# Каква е разликата между web server и web application ?

N Nadia

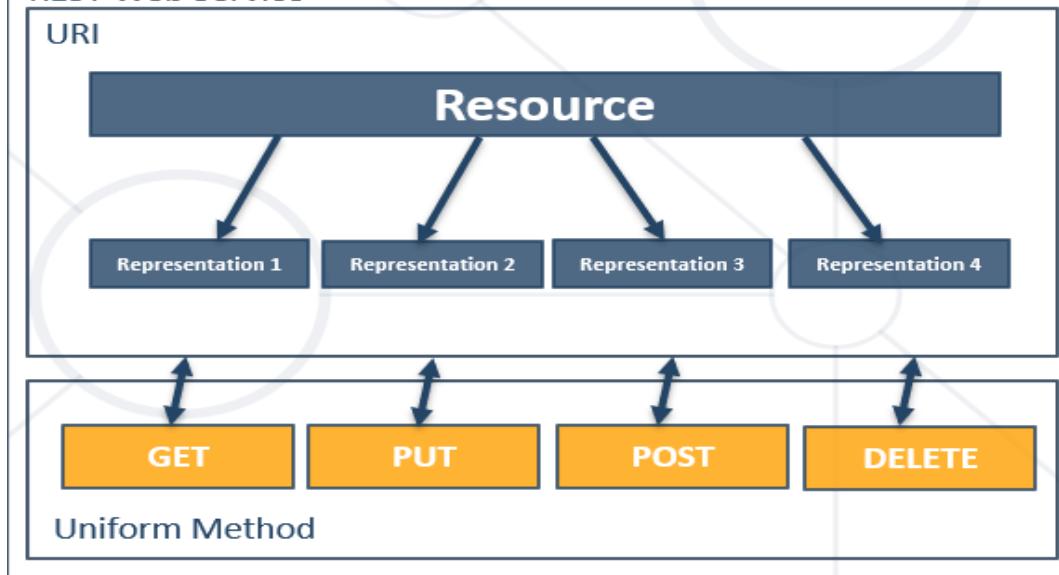
nginx  
Apache  
IIS  
node.js

CDN – content delivery network

HTTP 2 – дуплекс връзка, двупосочна

- **Representational State Transfer (REST)** – чрез HTTP работи на базата на определени принципи
  - Architecture for **client-server communication** over HTTP
  - Resources have **URI (Uniform Resource Identifier)** (address) – един и същи адрес за всички действия/команди, които извършваме върху този ресурс
  - Can be **created/retrieved/modified/deleted/etc.**
- RESTful API/RESTful Service – **уеб услуга, която работи на тези принципи**
  - Provides access to **server-side resources** via **HTTP** and **REST**

## REST Web Service



## REST Architectural Constraints

- REST defines **6 architectural constraints** which make any web service a true RESTful API (Application Programming language – нещо което може да ползваме наготово)
  - Client-server architecture – има една машина с роля клиент, и друга машина с роля сървър
  - Statelessness – няма състояние, на практика не е така – има кукита, и .т.н.
  - Cacheable
  - Layered system
  - Code on demand (optional)
  - Uniform interface
- Create a new post

**POST** | <http://some-service.org/api/posts>

- Get all posts / specific post

GET	<a href="http://some-service.org/api/posts">http://some-service.org/api/posts</a>
-----	-----------------------------------------------------------------------------------

GET	<a href="http://some-service.org/api/posts/17">http://some-service.org/api/posts/17</a>
-----	-----------------------------------------------------------------------------------------

- Delete existing post

DELETE	<a href="http://some-service.org/api/posts/17">http://some-service.org/api/posts/17</a>
--------	-----------------------------------------------------------------------------------------

- Replace / modify existing post

PUT/PATCH	<a href="http://some-service.org/api/posts/17">http://some-service.org/api/posts/17</a>
-----------	-----------------------------------------------------------------------------------------

PUT – изтрива стария запис и го заменя с нов

PATCH – редактира дадения запис

**Pagination** – подаване на информация от GET заявка по страници – за да не се трансферира цялата информация и да прави много трафик.

Прескочи първите 100 души, което при страница от 25 души са 4 страници

The screenshot shows a terminal window with a blue header. In the header, there is a user icon labeled "Anonymous" and some red handwritten-style numbers "100" and "5". To the right of these numbers, the text "offset = (page-1) \* size" is displayed in white. Below the header, the URL "https://reqres.in/" is shown in white. Underneath the URL, a red command is written: "data/movies?offset=100pageSize=25". At the bottom of the terminal window, another user icon labeled "Anonymous" is visible, followed by the text "При pagination, как се взима 7 страница ако са ни нужни първите 150 служители например ?" in white.

При **swapi.dev** има функция page

ions - Exercise: HTTP and REST Services - октомври 2021 - Виктор Костадинов

Now with The Force Awakens data!

Try it now!

https://swapi.dev/api/ people?page=2

request

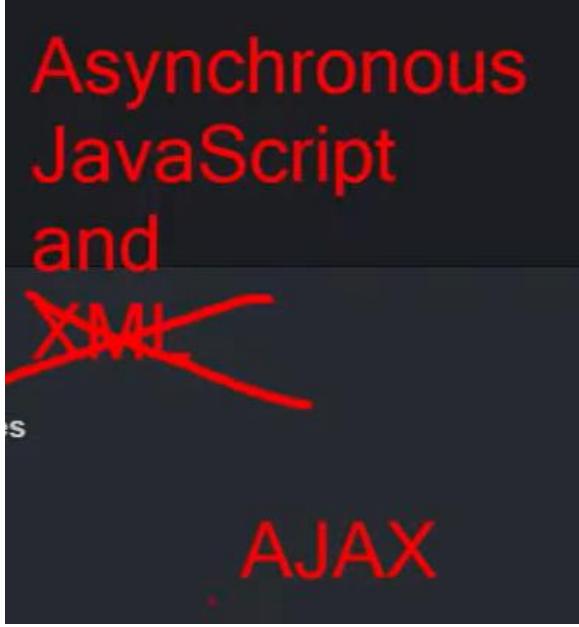
Need a hint? try [people/1/](#) or [planets/3/](#) or [starships/9/](#)

Result:

```
{
 "count": 82,
 "next": "https://swapi.dev/api/people/?page=3",
 "previous": "https://swapi.dev/api/people/?page=1",
 "results": [
 {
 "name": "Anakin Skywalker",
 "height": "188",
 "mass": "84",
 "hair_color": "blond",
 "skin_color": "fair",
 "eye_color": "blue",
 "birth_year": "41.9BBY",
 "gender": "male",
 "homeworld": "https://swapi.dev/api/planets/1/",
 "films": [
 "https://swapi.dev/api/films/4/",
 "https://swapi.dev/api/films/5/".
]
 }
]
}
```

YouTube

## AJAX



## 21.4. Accessing GitHub Through HTTP

### GitHub API

- List user's all public repositories:

GET	<a href="https://api.github.com/users/testnakov/repos">https://api.github.com/users/testnakov/repos</a>
-----	---------------------------------------------------------------------------------------------------------

- Get all commits from a public repository:

GET	<a href="https://api.github.com/repos/testnakov/softuniada-2016/commits">https://api.github.com/repos/testnakov/softuniada-2016/commits</a>
▪	Get all issues/issue #1 from a public repository
GET	<a href="/repos/testnakov/test-nakov-repo/issues">/repos/testnakov/test-nakov-repo/issues</a>

GET	<a href="/repos/testnakov/test-nakov-repo/issues/1">/repos/testnakov/test-nakov-repo/issues/1</a>
-----	---------------------------------------------------------------------------------------------------

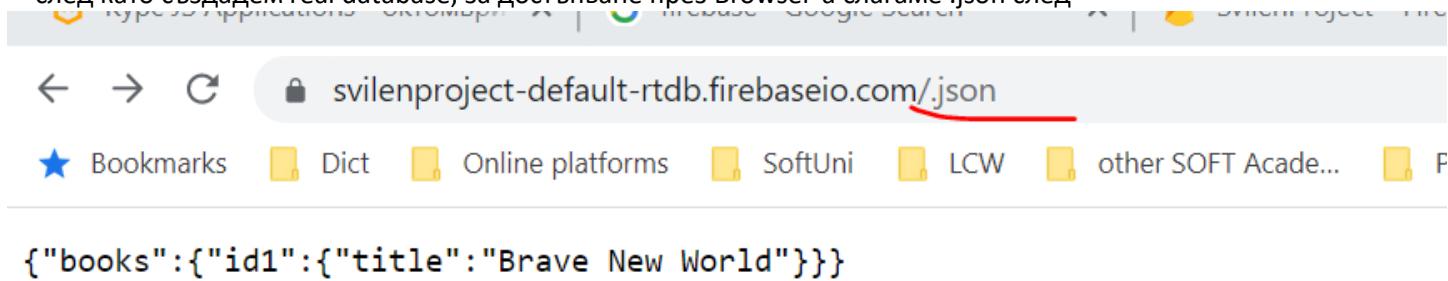
## 21.5. Popular Backend-as-a-Service (BaaS) Providers

- Web applications require a **back-end to store** information
  - User profiles, settings, content, etc.
- **Creating** a back-end can be very **time consuming**
- **Ready to use** back-end services are available (free trial): - за създаване на база данни
  - Firebase
  - Backendless
  - Back4App

And more

### Firebase

– след като създадем real database, за достъпване през Browser-а слагаме .json след



И в postman използваме пак /.json

GET

https://svilenproject-default-rtbd.firebaseio.com/books/.json

Params

Authorization ●

Headers (7)

Body

Pre-request Script

Tests

Settings

Body Cookies Headers (8) Test Results

Pretty

Raw

Preview

Visualize

JSON



```
1 {
2 "id1": {
3 "title": "Brave New World"
4 },
5 "id2": {
6 "aaa": "The Egyptian"
7 }
8 }
```

POST

https://svilenproject-default-rtbd.firebaseio.com/books/.json

Params

Authorization ●

Headers (9)

Body ●

Pre-request Script

Tests

Settings

none

form-data

x-www-form-urlencoded

raw

binary

GraphQL

JSON

```
1 {
2 "author": "Mika Waltari",
3 "title": "Brave New World"
4 }
```

Backendless

## New App

Give a name to your App

App name: myFirstApp

Parse Version: 4.2.0 - Parse Server 4.2.0

CREATE

or  
Find a template to clone

In order to **create a new collection (Class)** you can click on the **button "Create a Class"** (or "Create your first class")

myFirstApp

API Reference

Core

Database Browser **Create a class** NEW

Connections Index Manager Cloud Code Functions Webhooks Jobs Logs Config API Console

Publish on Hub NEW

Admin App

You have no classes yet

This is where you can view and edit your app's data

Create your first class

This will open a new window, where you need to choose the type of the collection and write its name.

## Add a new class

Create a new collection of objects.



What type of class do you need?

Custom

What should we call it?

Don't use any special characters, and start your name with a letter.

Countries

or

[Find a public dataset to connect](#)

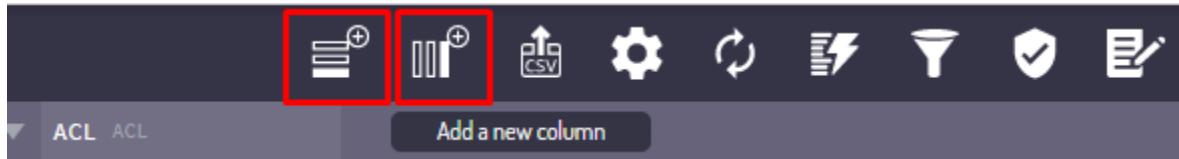
e.g. jobs, countries, industries, colors, zip codes and more...

[Cancel](#)

[Create class](#)

Now we have our first collection, which is empty at this time.

You can add new columns to this collection, and also new rows.



Let's try to add the columns: name, capital.

## Add a new column

Store another type of data in this class.

What type of data do you want to store?	String
What should we call it? Don't use any special characters, and start your name with a letter.	name
What is the default value? If no value is specified for this column, it will be filled with its default value.	Set a default value here
Is it a required field? When true this field must be filled when a new object is created.	No <input checked="" type="checkbox"/> Yes

[Never mind, don't](#)

[Add column](#)

And add some countries to this collection (rows). The first 4 columns are automatically filled; you don't need to fill them.

CLASS 11 OBJECT • PUBLIC READ AND WRITE ENABLED													
Countries							<a href="#">API Reference</a>	<a href="#">Video Tutorial</a>					
	objectId	String	updatedAt	Date	createdAt	Date	ACL	ACL	name	String	capital	String	<a href="#">Add a new column</a>
	K2llPjD7Ge		10 Feb 2021 at 11:...		10 Feb 2021 at 11:...		Public Read + Write		Bulgaria		Sofia		
<a href="#">+</a>													

Казва на нашето конкретно приложение/база данни адреса му.

1 object Student Public Read and Write enabled													
	objectId	String	updatedAt	Date	createdAt	Date	ACL	ACL	firstName	String	lastName	String	averageGrade
<input checked="" type="checkbox"/>	9xBzlKllHO		22 Nov 2021 at 21:...		22 Nov 2021 at 21:...		Public Read + Write		John		Snow		5.3
<a href="#">+</a>													

(1)

- [Security](#)
- [Browser As User](#)
- [Add a row](#)
- [Add a column](#)
- [Add a class](#)
- [Change pointer key](#)
- [Attach this row to relation](#)
- [Clone this row](#)
- [Delete this row](#)
- [Delete a column](#)
- [Delete all rows](#)
- [Delete this class](#)
- [Import](#)
- [Export](#)
- [Index Manager](#)
- [API Reference](#)



## Student-Demo-Up

**Introduction**

**Getting Started**

**Objects API**

**B4aVehicle Class API**

**Student Class API** (highlighted with a red circle)

- Creating Objects
- Reading Objects
- Updating Objects
- Deleting Objects

**User API**

## Student Class

Student is a custom class that was created and is specific for Student-Demo-Up. Please use the following documentation to learn how to perform CRUD (create, read, update and delete) operations to this specific class. A new endpoint was automatically generated at the address below to which you can send your requests:

<https://parseapi.back4app.com/classes/Student> (highlighted with a red circle)

The following fields are supported by this class' schema and can be used in the operations:

Name	Type	Example
firstName	String	"A string"

JavaScript
TypeScript
Android
Swift
Objective-C
PHP
.NET
cURL

**Example JSON:**

```
{
 "firstName": "A string",
 "lastName": "A string",
 "averageGrade": 1
}
```

Hello, svilkata\_sh

**NEW APP**

**JavaScript** **TypeScript** **Android** **Swift** **Objective-C** **PHP** **.NET** **cURL** (highlighted with a red circle)

Live

For

ant mode

no

ig multiple operators
 > likes between 50 and 400
 > ing with "Parse"

 ion-Id: Nw8icJOEcXZKC7JYw0pEY8YRdRzni5CEz0b7X5a" \
 -Key: TVMYThHlNBf929LdyY0oUmlenSr421jGSzFgsinm" \
 here={"likes":{"\$gte":50,"\$lte":400}, "title":{"\$regex": "^Parse"}}

**Student Class API** (highlighted with a red circle)

- Creating Objects
- Reading Objects
- Updating Objects
- Deleting Objects

**User API**

**POST**

**Headers**

X-Parse-Application-Id: uLOVcXGQ2hE6rY7DWYxVaaL8meuhhMWgpzJqMSF9

X-Parse-REST-API-Key: KUAm6nlldmjAvk6SA1EmkSgsaELd7vh8yAqNFL

Content-Type: application/json

**TRY ON JS FIDDLE**

**Example Output:**

```
{
 id: 'XKue915KBG',

```

В Postman слагаме адреса, и двата header-а ключ-стойност

GET https://parseapi.back4app.com/classes/Student Send

Params Authorization Headers (8) Body Pre-request Script Tests Settings Cookies

Headers 6 hidden

KEY	VALUE	DESCRIPTION	...	Bulk Edit	Presets
<input checked="" type="checkbox"/> X-Parse-Application-Id	uLOVcXGQ2hE6rY7DWYxVaaL8meuhhMWgpzJqMSF9...				
<input checked="" type="checkbox"/> X-Parse-REST-API-Key	KUAm6nlldmjAvk6SA1EmkSgsaELd7vh8yAqNFL...				
Key	Value	Description			

POST <https://parseapi.back4app.com/classes/Student>

Params Authorization Headers (10) Body Pre-request Script Tests Settings

Headers [8 hidden](#)

KEY	VALUE	DESCRIPTION
<input checked="" type="checkbox"/> X-Parse-Application-Id	uLOVcXGQ2hE6rY7DWYxVaaL8meuhhMWgpzJ...	
<input checked="" type="checkbox"/> X-Parse-REST-API-Key	KUAm6nlmjaAvk6SA1EmkSgsaELd7vh8yAq...	
Key	Value	Description

Body Cookies Headers (19) Test Results [Send](#) Status: 201 Created Time: 3

Pretty Raw Preview Visualize JSON [Raw](#)

```
1 {
2 "objectId": "jwPnLNP2Ic",
3 "createdAt": "2021-11-23T20:55:52.228Z"
4 }
```

POST <https://parseapi.back4app.com/classes/Student> [Send](#)

Params Authorization Headers (10) Body Pre-request Script Tests Settings Cookies

none  form-data  x-www-form-urlencoded  raw  binary  GraphQL JSON [Raw](#)

Beautiful

```
1 {
2 "firstName": "James",
3 "lastName": "Christopher",
4 "averageGrade": 5.7
5 }
```

Body Cookies Headers (19) Test Results [Save Response](#) Status: 201 Created Time: 311 ms Size: 1.1 KB

Pretty Raw Preview Visualize JSON [Raw](#)

```
1 {
2 "objectId": "jwPnLNP2Ic",
3 "createdAt": "2021-11-23T20:55:52.228Z"
4 }
```

PUT <https://parseapi.back4app.com/classes/Student/9xBzIKILHO> Send

Params Authorization Headers (10) **Body** Pre-request Script Tests Settings Cookies

none  form-data  x-www-form-urlencoded  raw  binary  GraphQL **JSON**

```

1 {
2 "firstName": "John",
3 "lastName": "Snow",
4 "averageGrade": 2.0
5 }
```

Изтриваме данните с id 9xBzIKILHO

DELETE <https://parseapi.back4app.com/classes/Student/9xBzIKILHO> Send

Params Authorization Headers (8) **Body** Pre-request Script Tests Settings Cookies

none  form-data  x-www-form-urlencoded  raw  binary  GraphQL

This request does not have a body

Body Cookies Headers (18) Test Results  Status: 200 OK Time: 217 ms Size: 990 B [Save Response](#)

Pretty Raw Preview Visualize **JSON**

```

1 {}
```

Лоша практика – URL не е криптиран никога, и тъпо там да слагаме като параметри пароли

Untitled Request

GET <https://parseapi.back4app.com/login?username=Peter&password=123456> Send

Params  Authorization Headers (8) Body Pre-request Script Tests Settings

Query Params

KEY	VALUE	DESCRIPTION
<input checked="" type="checkbox"/> username	Peter	
<input checked="" type="checkbox"/> password	123456	
Key	Value	Description

2. Всеки потребител може само да чете Quiz, но не може да редактира ако не е се е логнал. Ако се е логнал, може да редактира всеки quiz.

The screenshot shows the 'Edit Class Level Permissions' page for the 'Quiz' class. At the top, there are two rows of permissions:

Role	Read	Write	Add field
Public	<input checked="" type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
Authenticated	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input type="checkbox"/>

Below these rows is a text input field containing 'Role, User, or Pointer...'. In the top right corner of the main area, there is a gear icon.

3. Configure owner pointer - Всеки потребител може само да чете Quiz, но ако се е логнал може да редактира само quiz-те създадени от него.

The screenshot shows the 'Add a new column' configuration dialog. It has several sections:

- Add a new column**: Title of the dialog.
- Store another type of data in this class.**: Description.
- What type of data do you want to store?**: Set to 'Pointer'.
- Target class**: Set to 'User'.
- What should we call it?**: Placeholder text 'Give it a good name...'.
- What is the default value?**: Placeholder text 'Set a valid object ID here'.
- Is it a required field?**: A toggle switch set to 'Yes'.
- Action buttons at the bottom:**
  - 'Never mind, don't.'
  - 'Add column'
  - 'Add column & continue'

Edit Class Level Permissions

Permissions Simple Advanced

	Read			Write			Add
Public	Get <input checked="" type="checkbox"/>	Find <input checked="" type="checkbox"/>	Count <input checked="" type="checkbox"/>	Create <input type="checkbox"/>	Update <input type="checkbox"/>	Delete <input type="checkbox"/>	Add field <input type="checkbox"/>
Authenticated	Get <input checked="" type="checkbox"/>	Find <input checked="" type="checkbox"/>	Count <input checked="" type="checkbox"/>	Create <input checked="" type="checkbox"/>	Update <input type="checkbox"/>	Delete <input type="checkbox"/>	Add field <input type="checkbox"/>
owner Pointer<_User> Only users pointed to by this field	Get <input checked="" type="checkbox"/>	Find <input checked="" type="checkbox"/>	Count <input checked="" type="checkbox"/>	Create <input checked="" type="checkbox"/>	Update <input checked="" type="checkbox"/>	Delete <input checked="" type="checkbox"/>	Add field <input type="checkbox"/> <span style="color: red;">X</span>
Role, User, or Pointer...							

[Learn more about CLPs and app security](#)

[Cancel](#) [Save CLP](#)

data.js

```
function createPointer(name, id) {
 return {
 __type: 'Pointer',
 className: name,
 objectId: id
 };
}

function addOwner(object) {
 const userId = getUserData().userId;
 object.owner = createPointer('_User', userId);
}

export async function createQuiz(quiz) {
 addOwner(quiz);
 return await api.post(host + '/classes/Quiz', quiz)
}
```

Objects | X Public Read enabled

Quiz

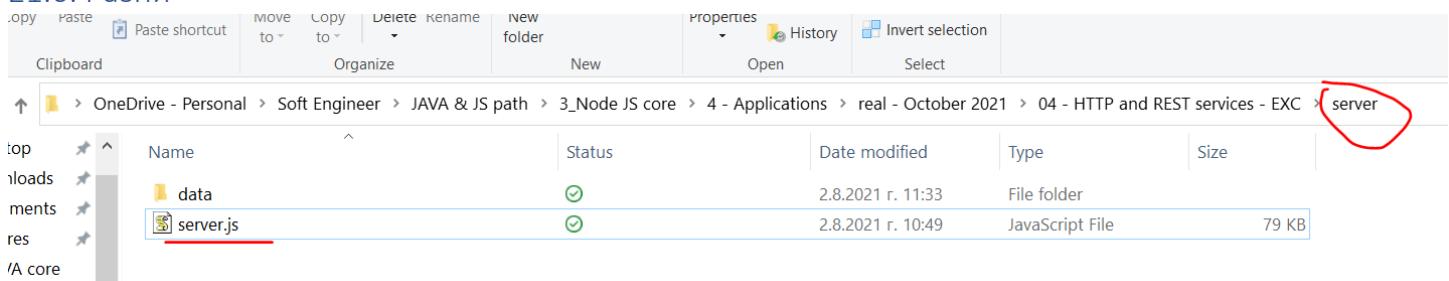
createdAt	Date	ACL	title	String *	topic	String *	questionCount	Number	owner	Pointer <_User>	Actions
' May 2022 at 23:3...		Public Read + Write	John's Quiz		Test Qqqquiz		0		S4qaVaaJl6	<span style="color: red;">V</span>	<span style="color: blue;">🔗</span>
' May 2022 at 22:5...		Public Read + Write	First Quiz		Programming		0		(undefined)		

#### 4. Setup steps

1. x Initialize project
2. x Copy / api
3. x Create Back4Up app
4. View documentation on Users
5. x Adjust api.js to include AppID, APIKey, correct headers, register/login/logout routes and bodies

6. x Test register/login/logout
7. Examine Database Browser, create Quiz collection
8. x Create CRUD functions for Quiz collection
9. x Test Quiz collection, confirm it is public for read/write, NO Add field CLP
10. View documentation on security, ACL, CLP
11. x Configure public read, authenticated write, NO add field CLP
12. x Test read/write
13. x Configure owner pointer
14. x Adjust data.js to include owner pointer on create - { \_\_type: 'Pointer', className: 'User'}
15. x Test owner protection
16. Add query to include owner on GET
17. Create Question collection, configure CLP, owner pointer, Quiz pointer
18. Create CRUD functions for Question collection, with owner protection
19. Test Question collection
20. Proceed with implementation of views

## 21.6. Разни



MINGW64:/c/Users/svilk/OneDrive/Soft Engineer/JAVA & JS path/3\_Node JS core/4 - Applications/real - October 2021/04 - HTTP and REST services - EXC/server

```

svilk@DESKTOP-SVILEN MINGW64 ~/OneDrive/Soft Engineer/JAVA & JS path/3_Node JS core/4 - Applications/real - October 2021/04 - HTTP and REST services - EXC/server
$ node server.js
Server started on port 3030. You can make requests to http://localhost:3030/
Admin panel located at http://localhost:3030/admin

svilk@DESKTOP-SVILEN MINGW64 ~/OneDrive/Soft Engineer/JAVA & JS path/3_Node JS core/4 - Applications/real - october 2021/04 - HTTP and REST services - EXC/server
$ node server.js
Server started on port 3030. You can make requests to http://localhost:3030/
Admin panel Located at http://localhost:3030/admin
<< GET /admin
<< GET /admin/
<< GET /util/throttle
<< GET /data
<< GET /favicon.ico
serving favicon...
<< GET /util/throttle
<< GET /data/recipes
<< GET /util/throttle

```

C Ctrl + C спираме изпълнението на сървъра - ^C

## Notable well-known port numbers

Number	Assignment
20	<a href="#">File Transfer Protocol</a> (FTP) Data Transfer
21	<a href="#">File Transfer Protocol</a> (FTP) Command Control
22	<a href="#">Secure Shell</a> (SSH) Secure Login
23	<a href="#">Telnet</a> remote login service, unencrypted text messages
25	<a href="#">Simple Mail Transfer Protocol</a> (SMTP) email delivery
53	<a href="#">Domain Name System</a> (DNS) service
67, 68	<a href="#">Dynamic Host Configuration Protocol</a> (DHCP)
80	<a href="#">Hypertext Transfer Protocol</a> (HTTP) used in the <a href="#">World Wide Web</a>
110	<a href="#">Post Office Protocol</a> (POP3)
119	<a href="#">Network News Transfer Protocol</a> (NNTP)
123	<a href="#">Network Time Protocol</a> (NTP)
143	<a href="#">Internet Message Access Protocol</a> (IMAP) Management of digital mail
161	<a href="#">Simple Network Management Protocol</a> (SNMP)
194	<a href="#">Internet Relay Chat</a> (IRC)

Заменяме :id с конкретен номер на автобусната спирка в случая.

- **GET:** <http://localhost:3030/jsonstore/bus/businfo/:stopID>

I will receive a JSON object in the format:

```
stopID: {
 name: stopName,
 uses: { busId, time, ... }
}
```

1327

:id

:collection

Алтернативи на Postman:

<https://hoppscotch.io/>

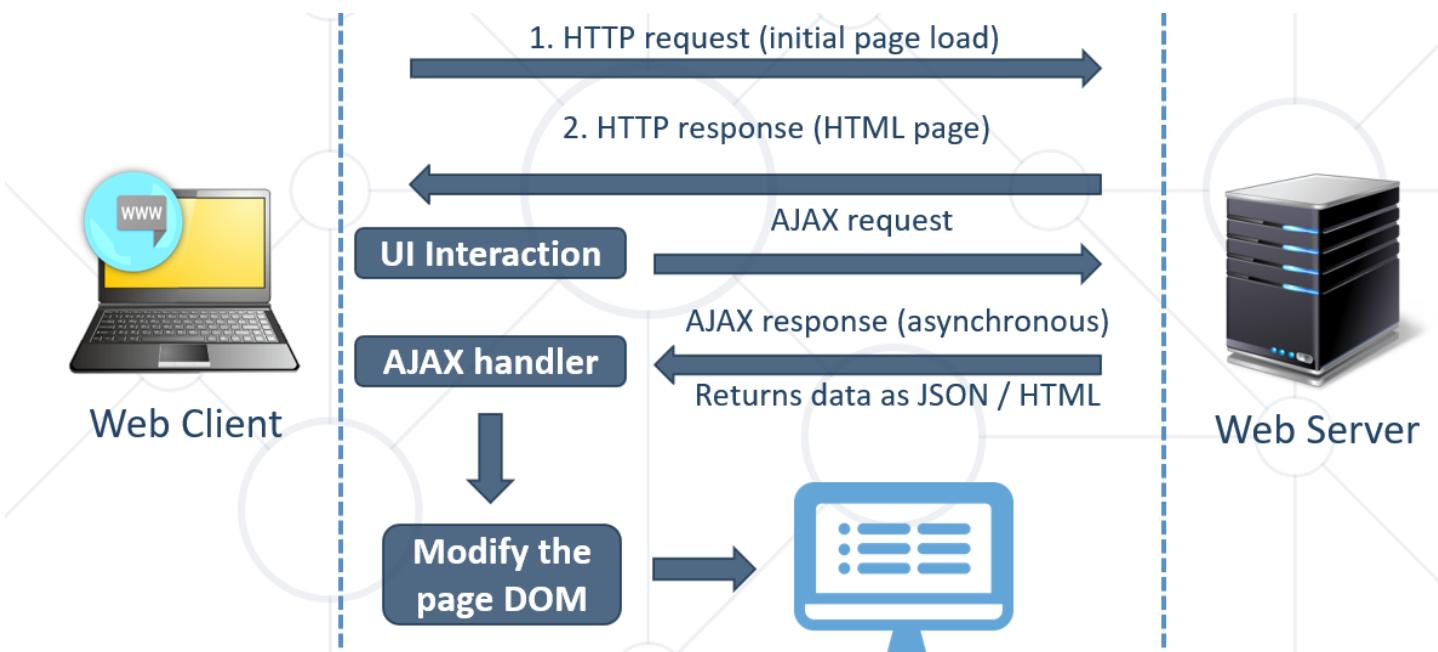
<https://insomnia.rest/>

## 22. Asynchronous Programming and Promises

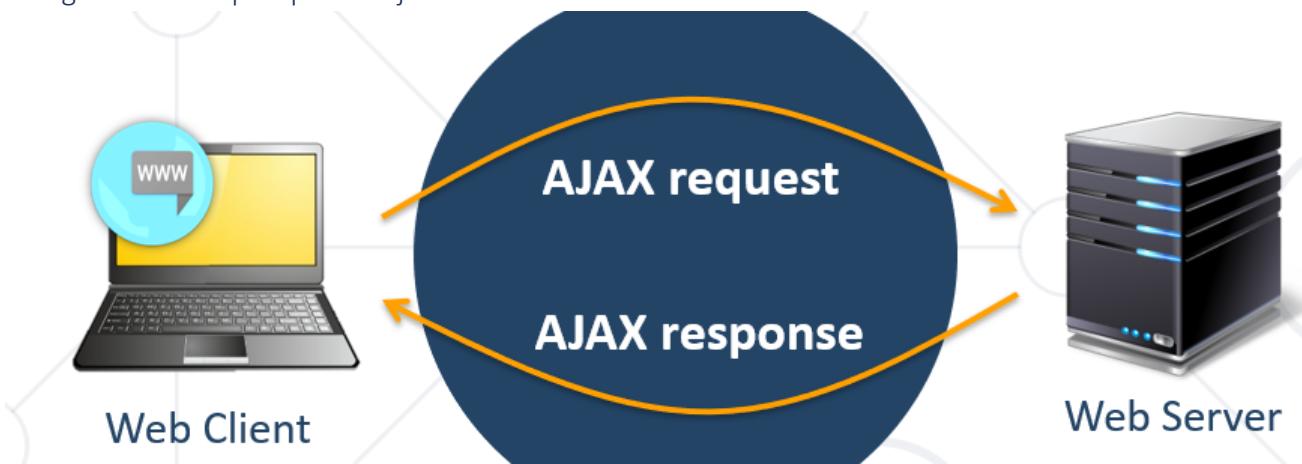
### 22.1. AJAX - Asynchronous JavaScript and XML JSON

What is AJAX? = Asynchronous JavaScript and XML JSON

- **Asynchronous JavaScript And XML**
  - Background loading of **dynamic content/data**
  - Load data from the Web server and **render** it
- Two types of AJAX
  - **Partial page rendering**
    - Load HTML fragment + show it in a <div>
  - **JSON service**
    - Loads JSON object and displays it



Using the XMLHttpRequest Object



#### XMLHttpRequest – Standard API for AJAX – GET Заявка

```

<button id = "load">Load Repos</button>
<div id="res"></div>

let button = document.querySelector("#load");
button.addEventListener('click', function loadRepos() {
 let url = 'https://api.github.com/users/testnakov/repos';
 const httpRequest = new XMLHttpRequest(); // класа името е legacy, реално ползваме JSON
 httpRequest.addEventListener('readystatechange', function () {
 if (httpRequest.readyState == 4 && httpRequest.status == 200) {
 document.getElementById("res").textContent = httpRequest.responseText;
 }
 });
});

```

```

httpRequest.open("GET", url); // GET Заявката
httpRequest.send(); //ИЗПРАЩАМЕ Заявката
});

```

## 22.2. Synchronous vs Asynchronous

### Asynchronous Programming in JS

- Not the same thing as **concurrent** or **multi-threaded**
- There can be **asynchronous code**, but it is **generally single-threaded**
- Structured using **callback functions**
- In current versions of JS there are:

### Callbacks

```

<button id = "load">Load Repos</button>
<div id="res"></div>

let button = document.querySelector("#load");
button.addEventListener('click', function loadRepos() {
 let url = 'https://api.github.com/users/testnakov/repos';
 const httpRequest = new XMLHttpRequest();
 httpRequest.addEventListener('readystatechange', function () {
 if (httpRequest.readyState == 4 && httpRequest.status == 200) {
 document.getElementById("res").textContent = httpRequest.responseText;
 }
 });
 httpRequest.open("GET", url);
 httpRequest.send();
});

```

callback event listener

- Function **passed** into another function as an **argument**
- Then **invoked** inside the outer function to complete some kind of routine or action

```

function running() {
 return "Running";
}

function category(run, type) {
 console.log(run() + " " + type);
}

category(running, "sprint"); //Running sprint

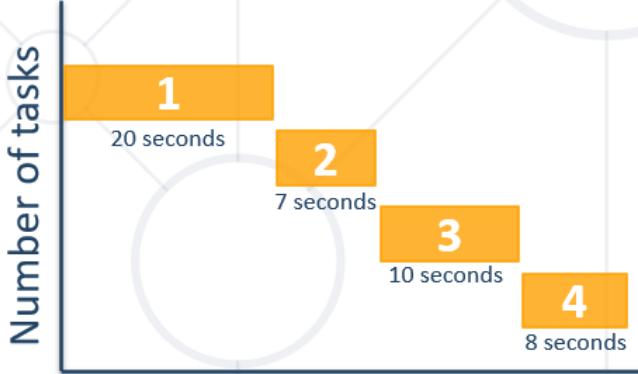
```

### Promises

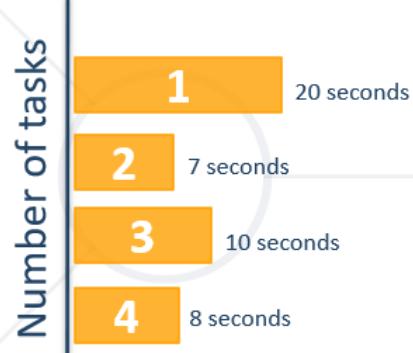
### Async Functions

- Runs several tasks (pieces of code) in parallel, **at the same time**

# Synchronous



# Asynchronous



## 22.3. Promises

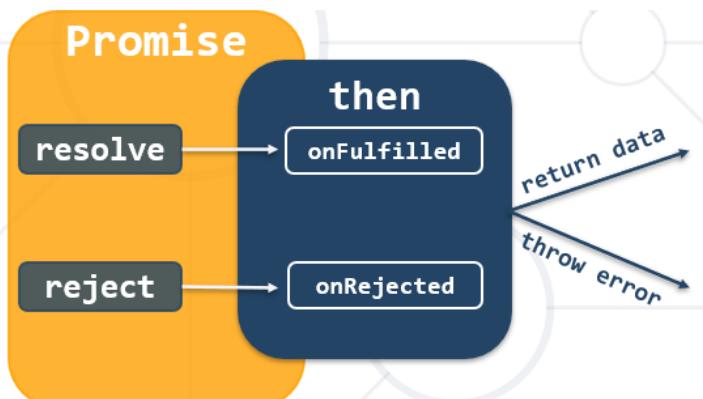
Objects Holding Asynchronous Operations

What is a Promise?

- A promise is an **asynchronous action** that **may complete** at some point and **produce a value**
- States:
  - **Pending** - operation still running (unfinished)
  - **Fulfilled** - operation finished (the result is available)
  - **Failed** - operation failed (an error is present)
- Promises use the **Promise** object
- Промиса се зарежда в паметта, и можем вече да си го викаме този промис с callback

```
new Promise(executor); new Promise(executor);
```

Promise Flowchart



What is Fetch?

- The **fetch()** method allows making network requests
- It is similar to **XMLHttpRequest** (XHR). The main **difference** is that the **Fetch API of the browser**:
  - Uses **Promises**

- Enables a **simpler** and **cleaner** API
- Makes code more readable and maintainable

```
fetch('./api/some.json')
 .then(function(response) {...} или response => response.json())
 .then(function(data) {...} или data => console.log(data))
 .catch(function(err) {...})

<body>
 <button>Click me</button>
 <script>
 document.querySelector('button').addEventListener('click', makeRequest);

 function makeRequest() {
 fetch('https://swapi.dev/api/planets/1/')
 .then(response => response.json()) //взема следващата част от заявката = дай ми header-а. Response е header-а на HTTP заявката - we get promise here
 .then(data => console.log(data)); //взема следващата част от заявката = дай ми body-то. data е информацията от HTTP заявката . - we get data here

 }
 </script>
</body>
```

### Basic Fetch Request

- The response of a **fetch()** request is a **Stream** object
- The **reading** of the stream happens **asynchronously**
- When the **json()** method is called, a **Promise** is **returned**
  - The **response status** is checked (should be **200**) before parsing the response as **JSON**

```
if (response.status !== 200) {
 // handle error
}

response.json()
 .then(function (data) { console.log(data) })
```

### GET Request

- **Fetch API** uses the **GET** method so that a direct call would be like this

```
fetch('https://api.github.com/users/testnakov/repos')
 .then((response) => response.json())
 .then((data) => console.log(data))
 .catch((error) => console.error(error))
```

### POST Request

- To make a **POST** request, we can set the **method** and **body** parameters in the **fetch()** options

```

fetch('/url', {
 method: 'post',
 headers: { 'Content-type': 'application/json' },
 body: JSON.stringify(data),
})

<script>
 async function postData() {
 const url = 'http://localhost:3030/jsonstore/phonebook';
 const data = {
 person: "Zvezdi",
 phone: "+1-000-1111"
 };

 const options = {
 method: 'post',
 headers: { 'Content-type': 'application/json' },
 body: JSON.stringify(data)
 }

 fetch(url, options);
 }
</script>

```

Update(PUT) Request

```

<script>
 async function updateData(id, data) {
 const url = 'http://localhost:3030/jsonstore/phonebook' + id;

 const options = {
 method: 'put',
 headers: { 'Content-type': 'application/json' },
 body: JSON.stringify(data)
 }

 const response = await fetch(url, options);
 const result = await res.json();

 return result;
 }
</script>

```

Delete(DELETE) Request

```

<script>
 async function updateData(id) {
 const url = 'http://localhost:3030/jsonstore/phonebook' + id;

 const options = {

```

```

 method: 'delete',
 }

 const response = await fetch(url, options);
 const result = await res.json();

 return result;
}
</script>

```

## Body Methods

- **clone()** create a clone of the response
- **json()** resolves the promise with JSON
- **redirect()** create new promise but with different URL
- **text()** resolves the promise with string
- **arrayBuffer()** resolve body with ArrayBuffer
- **blob()** resolve body with Blob (file, image, etc.)
- **formData()** resolve body with FormData

## Response Types

- **basic** - normal, same origin response
- **cors** - response was received from a valid cross-origin request
- **error** - error network
- **opaque** - Response for "no-cors" request to cross-origin resource
- **opaqueredirect** - the fetch request was made with **redirect: "manual"**

## CORS - Access-Control-Allow-Origin

```

Access-Control-Allow-Origin: *
Access-Control-Allow-Origin: <origin>
Access-Control-Allow-Origin: null

```

## Chaining Promises

- When working with a JSON API, you can:
  - Define the **status** and **JSON parsing** in **separate functions**
  - The functions **return promises** which can be **chained**

```

fetch('users.json')
 .then(status) //първият then обработва статуса на response-a
 .then(json) //първият then обработва главата на response-a
 .then(function (data) {... })
 .catch(function (error) {... });

```

## Вариант 1

```
function loadRepos() {
```

```

const url = 'https://api.github.com/users/testnakov/repos';

fetch(url)
.then(response => response.json()) //първият then обработва главата на response-а
.then(data => console.log(data));
}

```

### **Вариант 2**

```

function loadRepos() {
 const url = 'https://api.github.com/users/testnakovfdas/repos';

 fetch(url)
 .then(res => {
 if (res.ok == false) {
 throw new Error(` ${res.status} ${res.statusText}`);
 }

 return res.json();
 })
 .then(data => console.log(data))
 .catch(error => console.log(error));
}

```

### **Вариант 3**

```

function loadRepos() {
 const username = document.getElementById('username').value;
 const url = `https://api.github.com/users/${username}/repos`;
 const list = document.getElementById('repos');

 fetch(url)
 .then(res => {
 if (res.ok == false) {
 throw new Error(` ${res.status} ${res.statusText}`);
 }

 return res.json();
 })
 .then(handleResponse)
 .catch(handleError);

 function handleResponse(data) {
 list.innerHTML = '';

 for(let repo of data){
 const liElement = document.createElement('li');
 liElement.innerHTML = `
 ${repo.full_name}
 `;
 }
 }
}

```

```

 // console.log(repo.full_name, repo.html_url);
 list.appendChild(liElement);
 }
}

function handleError(error) {
 list.innerHTML = '';
 list.textContent = `${error.message}`;
}
}

```

## 22.4. Async / Await

### Async Functions

- Returns a **promise**, that can await other promises in a way that **looks synchronous**
- Operate **asynchronously** via the event loop
- Contains an **await** expression that:
  - Is **only valid** inside **async functions**
  - **Pauses** the execution of that function
  - Waits for the Promise's **resolution**

**Await замества всички then-ове изобщо. Колкото then, толкова броя await трябва да имаме**

```

<body>
 <button>Click me</button>
 <script>
 document.querySelector('button').addEventListener('click', makeRequest);

 async function makeRequest() { //автоматично връща promise тази асинхронна функция
 try {
 const response = await fetch('https://swapi.dev/api/planets/1/');
 if (response.ok == false) {
 throw new Error(` ${response.status} ${response.statusText}`);
 }

 const data = await response.json();
 console.log(data);
 } catch (error) {

 }
 }

```

Вариантът без async и без await

```

// function makeRequest() {
// fetch('https://swapi.dev/api/planets/1/')
// .then(response => response.json())
// .then(data => console.log(data));
// }

```

```
</script>
</body>
```

- Do not confuse **await** with **Promise.then()**
  - **await** is always used for a **single promise**
  - To **await two or more promises in parallel**, use **Promise.all()**
- If a promise resolves normally, then **await** promise **returns the result**
- In case of a rejection, it **throws an error**

Async/Await vs Promise.then

## ■ Promise.then

```
function logFetch(url) {
 return fetch(url)
 .then(response => {
 return response.text()
 })
 .then(text => {
 console.log(text);
 })
 .catch(err => {
 console.error(err);
 });
}
```

## ■ Async/Await

```
async function logFetch(url) {
 try {
 const response =
 await fetch(url);
 console.log(
 await response.text()
);
 } catch (err) {
 console.log(err);
 }
}
```

### Пример:

Fetch връща promise

Await превръща promise-а **в данни**/в response-а, който е вътре в него.

response.json() връща promise на бодито

Await превръща promise-а с бодито **в данни/самото body**.

```
const response = await fetch('https://swapi.dev/api/planets/1/');
const data = await response.json();
```

### Error Handling

```
async function f() {
 try {
 let response = await fetch();
 let user = await response.json();
 } catch (err) {
 // catches errors both in fetch and response.json
 alert(err);
 }
}
```

```
async function f() {
 let response = await fetch();
}
// f() becomes a rejected promise
f().catch(alert);
```

## Sequential Execution

- To execute different promise methods **one by one**, use **Async /Await**

## Concurrent Execution

`Promise.all()`

`await Promise.all([fetch('v1'), fetch('v2'), fetch('v3')]);` - три промиса се пакетират на само един промис, и на обединения промис му даваме await.

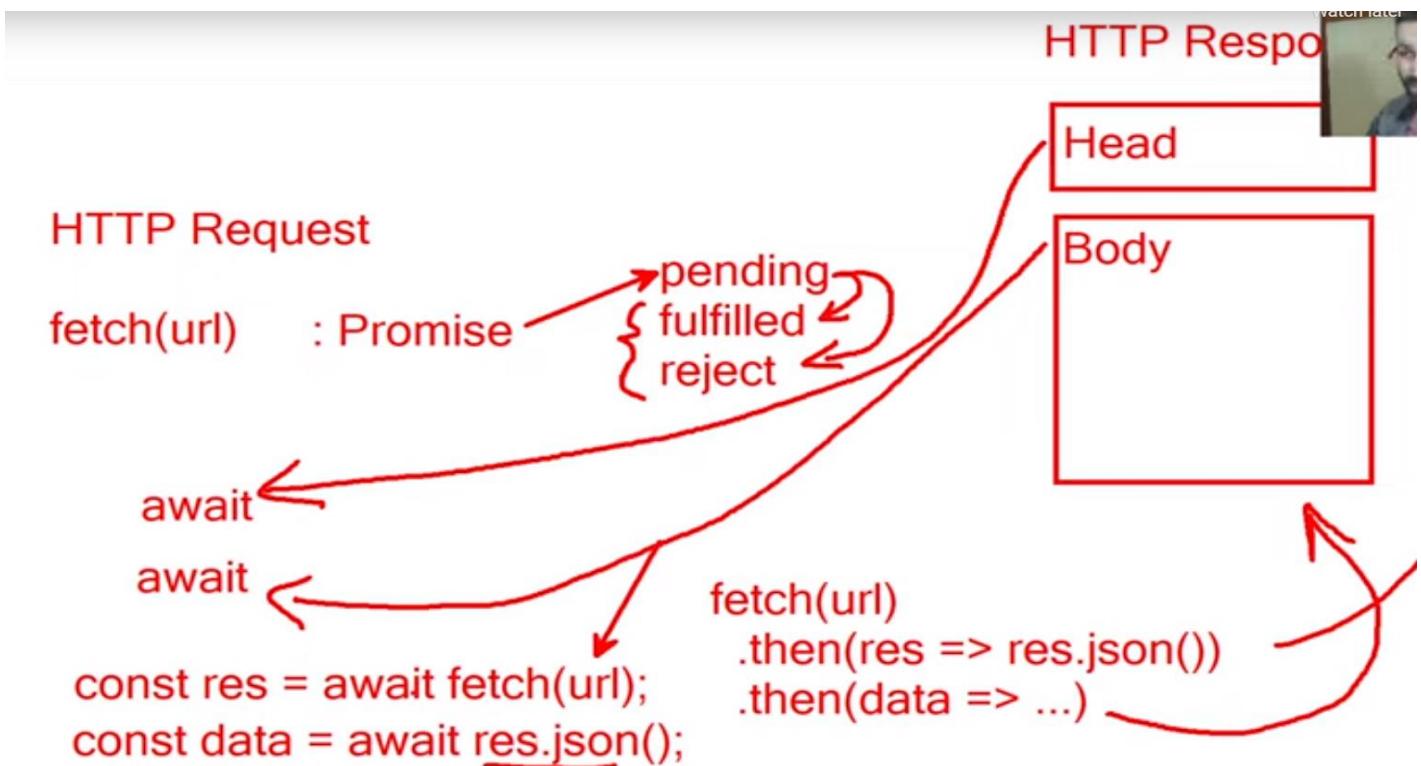
```
const v1 = await fetch('v1');
const v2 = await fetch('v2');
const v3 = await fetch('v3');
```

## Асинхронна заявка връща Promise

**Async** functions contain an **await** expression

**Yields the execution**

**Waits for the Promise's resolution**



## `Promise.race()`

The `Promise.race()` method returns a promise that fulfills or rejects as soon as one of the promises in an iterable fulfills or rejects, with the value or reason from that promise.

# 23. Data and Authentication

## 23.1. Remote Storage

- The client can **send data** to the server, usually via **POST request**

```
const data = {title: 'First Post', content: 'Hello, Server!'};
fetch('/articles', {
 method: 'post',
 headers: { 'Content-type': 'application/json' },
 body: JSON.stringify(data),
});
```

- This allows:
  - Specialized requests, such as **filtering** collections
  - **Permanent storage** and **sharing** of content

### Request Options

- Provide an **options** object to **Fetch API** to send data:
  - **method** – can be **POST**, **PUT**, **PATCH** or **DELETE**
  - **body** – contains the **data** to be sent, usually as **JSON** string
  - **headers** – common headers include:
    - **Content-Type** specifies the **format** of the data (**manual**)
    - **Content-Length** specifies the **size** of the data (**automatic**)
    - **Cookie** can be used with **authentication** (**automatic**)
    - Custom **authorization** headers (**manual**)

## 23.2. Database Principles

### Backend As a Service

- Solutions that provide **pre-built**, **cloud** hosted components for developing **application backends**
- Reduce the **time** and **complexity** required
- Allow developers to focus on **core features** instead of low-level tasks
- Types:
  - Cloud BaaS
  - Open-source BaaS

## of low-level tasks

- Types:

- Cloud BaaS
- Open-source BaaS



5:25



### Relational Databases

- Represent and store **data** in tables and rows
- Use **Structured Querying Language (SQL)**
- Allows you to link information from different tables through the use of foreign keys (or indexes)

### Non-Relational Databases

- No-SQL databases
- More **flexibility** and **adaptability**
- Allow us to **store unstructured data** in a single document (*not a good idea*)
- Additional processing effort and **more storage** as the document sizes grow



### Relational and Non-Relational Pros

- Relational
  - Work with **structured data**
  - They support **ACID** transactional consistency and support "**joins**"
  - Built-in **data integrity** and a large eco-system
  - Relationships in this system have **constraints**
  - Limitless **indexing**
- Non-Relational
  - They **scale out horizontally**
  - Work with **unstructured** and semi-structured **data**
  - Schema-free or Schema-on-read options
  - **High availability**
  - Many are **open source** and so "free"

### Working with NoSQL Collections

- Records in a database have **unique identification keys**
  - New records are usually **assigned** and Id **automatically**

- This allows a record to be **retrieved directly**
- **Keys** can be used to create a **relationship** between records
- It's best to impose a **structure** on all records
  - Every entry has the **same properties**
- **De-normalize** data
  - E.g., article comments can be stored **inside** the article – вложени записи

### 23.3. Handling Forms

#### HTML Form Standard

- The **<form>** element groups many **<input>** fields
  - Attribute **method** specifies which **HTTP method** to use
  - Attribute **action** specifies to which **URL** the request is sent
- On **submit**, the browser **sends all values** to the server
  - Every input is identified by its **name attribute**

#### Handling Submit Request

- Browser form submission causes the **page to reload**
  - Our application will be **closed** or **restarted**
- The **submit event** can be **intercepted**
- A **fetch request** can be made using the **input values**

```
const formElement = document.querySelector('form');
formElement.addEventListener('submit', event => {
 event.preventDefault();
 // collect values and send via fetch
});
```

#### Working with FormData

- The **FormData** object **automatically serializes** all input values of the DOM elements
  - No need to select them manually
  - Вместо да селектираме нещата от DOM обекта, за по-бърза/лесна работа
  - Реално взема данните от всеки DOM обект, най-вече съобщенията

```
formElement.addEventListener('submit', event => {
 event.preventDefault();
 const data = new FormData(formElement);
 const email = data.get('email'); // Read single value
 const entries = [...data.entries()]; // Get array of values
});
```

#### Пример:

```
<body>
<form method="POST" action="http://localhost:3030/jsonstore/phonebook">
```

```
<label>Person: <input type="text" name="person" /> </label>
<label>Phone: <input type="text" name="phone" /> </label>

<input type="submit" value="Create Record" />
</form>

<script>
 const form = document.querySelector('form');
 form.addEventListener('submit', onSubmit);
 const personInput = document.querySelector('[name="person"]');
 const phoneInput = document.querySelector('[name="phone"]');

 async function onSubmit(event) {
 event.preventDefault();
 const data = new FormData(form);

 // const person = personInput.value.trim();
 // const phone = phoneInput.value.trim();
 const person = data.get('person').trim();
 const phone = data.get('phone').trim();

 const record = { person, phone };

 // console.log(person, phone);
 const result = await postData(record);
 personInput.value = '';
 phoneInput.value = '';
 alert('Record created');
 }

 async function getData() {
 const url = 'http://localhost:3030/jsonstore/phonebook';

 const res = await fetch(url);
 const data = await res.json();

 return data;
 }

 async function postData(data) {
 const url = 'http://localhost:3030/jsonstore/phonebook';

 const options = {
 method: 'post',
 headers: { 'Content-type': 'application/json' },
 body: JSON.stringify(data)
 }

 try {
 const res = await fetch(url, options);
 }
 }
</script>
```

```

 if (res.ok != true) {
 throw new Error(res.statusText);
 }

 const result = await res.json();
 return result;
 } catch (error) {

 }
}
</script>
</body>

```

## 23.4. Authentication - working with user Credentials

### Authentication and Authorization

- **Authentication**
  - The process of verifying the identity of a user or computer
  - Questions: "**Who are you?**", "**How you prove it?**"
  - Credentials can be password, smart card, external token, etc.
- **Authorization**
  - The process of determining what a user is permitted to do on a computer or network
  - Questions: "**What are you allowed to do?**", "**Can you see this page?**"

### Common Authentication Techniques

#### *HTTP Basic Authentication – credentials with every request*

- Username and password sent in a **request header**:

**Authorization: Basic dXNlcm5hbWU6cGFzc3dvcmQ=**

btoa() - криптира

atobe() - декриптира

#### *Cookie – upon login, server returns authentication cookie*

#### *Token-based – upon login, server returns signed token*

Usually sent in a **request header** (name varies):

**Auth-Token: d50d5f194848683ec68d2d0c4595128b146551249...**

- Other methods: One Time passwords, Oauth, OpenID, etc.

Влизане с външен доставчик – Google, facebook, github, или други

### Registration Request

```
<form method="POST" action="/users/register">
 <input type="text" name="email" />
 <input type="password" name="password" />
 <input type="password" name="repass" />
 <input type="submit" value="Register" />
</form>

async function onRegister(ev) {
 const response = await fetch('/users/register', {
 method: 'post',
 headers: { 'Content-Type': 'application/json' },
 body: JSON.stringify(ev.formData)
 })
}
```

### Register Request and Login Request

```
<body>

 <h2>Register</h2>
 <form id="register-form">
 <label>Email: <input type="text" name="email" /> </label>
 <label>Password: <input type="password" name="password" /> </label>
 <label>Repeat: <input type="password" name="repas" /> </label>
 <input type="submit" value="Register" />
 </form>

 <h2>Login</h2>
 <form id="login-form">
 <label>Email: <input type="text" name="email" /> </label>
 <label>Password: <input type="password" name="password" /> </label>
 <input type="submit" value="Login" />
 </form>

 <script>
 const registerForm = document.getElementById('register-form');
 registerForm.addEventListener('submit', onRegister);

 async function onRegister(event) {
 event.preventDefault();
 const formData = new FormData(registerForm);
 const url = 'http://localhost:3030/users/register';

 const email = formData.get('email').trim();
 const password = formData.get('password').trim();
 const repass = formData.get('repas').trim();
 }
 </script>
```

```

const response = await fetch(url, {
 method: 'post',
 headers: {
 'Content-type': 'application/json',
 },
 body: JSON.stringify({
 email, password
 })
});

const result = await response.json();

const token = result.accessToken;
sessionStorage.setItem('myToken', token); //като затворим браузъра/таба, ще
се изтрие
localStorage.setItem('myToken', token); //има по-дълъг живот от sessionStorage - пази го и при рестартиране на компютъра

```

Потребителят като се е логнал веднъж, много по-логично е различните табове да не му искат всеки път парола, та затова използваме localStorage.

При sessionStorage, нов таб на същия сайт иска наново парола.  
sessionStorage.clear() – занулява данните в текущата сесия

```

 console.log(result);
}

const loginForm = document.getElementById('login-form');
loginForm.addEventListener('submit', onLogin);

async function onLogin(event) {
 const url = 'http://localhost:3030/users/login';

 event.preventDefault();
 const formData = new FormData(loginForm);

 const email = formData.get('email').trim();
 const password = formData.get('password').trim();

 const response = await fetch(url, {
 method: 'post',
 headers: {
 'Content-type': 'application/json',
 },
 body: JSON.stringify({
 email, password
 })
 });

 const result = await response.json();

 const token = result.accessToken;

```

```

 sessionStorage.setItem('myToken', token);

 console.log(result);
 }

</script>
</body>

```

## Handling Authentication Token

- Upon **successful login**, the server returns authentication **token**
  - This token must be **attached** to every **subsequent request**
  - **Save it using sessionStorage:**

```

const authToken = response.authToken;
sessionStorage.setItem('authToken', authToken);

```

- **Send it in a request header:**

```

fetch('/articles', {
 method: 'get',
 headers: { 'X-Authorization': authToken }
});

```

```

//Оторизирана заявка GET
async function getData() {
 const url = 'http://localhost:3030/jsonstore/phonebook';
 const options = { headers: {} };

 const token = sessionStorage.getItem('token');
 if (token !== null) {
 options.headers['X-Authorization'] = token;
 }

 const res = await fetch(url);
 const data = await res.json();

 return data;
}

```

**Network** грешките хвърлят exception, и реално се спира JS кода изпълнението на останалата част от дадената функция.

## Data Ownership and Authorization

- Most APIs will **record the data's author**

- Stored as **ownerId**, **creator** or similarly named **property**
- Can be used to e.g., identify an article's or comment's author
- Depending on the service's **access rules**, only the author (and possibly administrators) **can modify** their records
- Display **edit controls** for records owned by the **current user**
  - Note that visibility **does not** provide security – this is done **on the server**, using access rules

## 23.5. Обобщение

Генерира се първоначално token, с който след това post-ваме и get-ваме информация от сървъра под определ username и парола.

В реална ситуация, живоът на един token е ограничен. След малко време, ще иска да въведем нова/наново парола и да се генерира нов токен.

Нещата в JS можем да ги правим и на Postman реално.

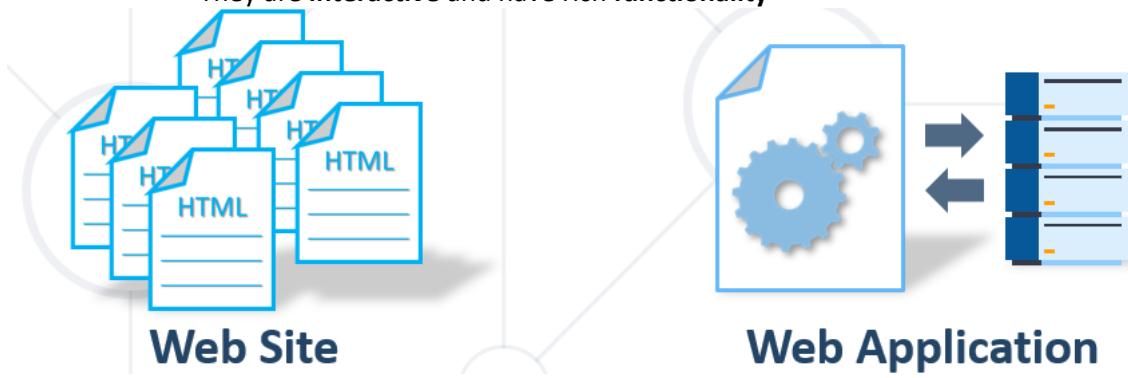
Четем документацията на сървъра, за да знаем в дадени ситуации какъв тип грешка хвърля сървъра – 200, 201, 202, 204....

## 24. Single Page Applications

### 24.1. Web Application Concepts

#### Sites and Applications

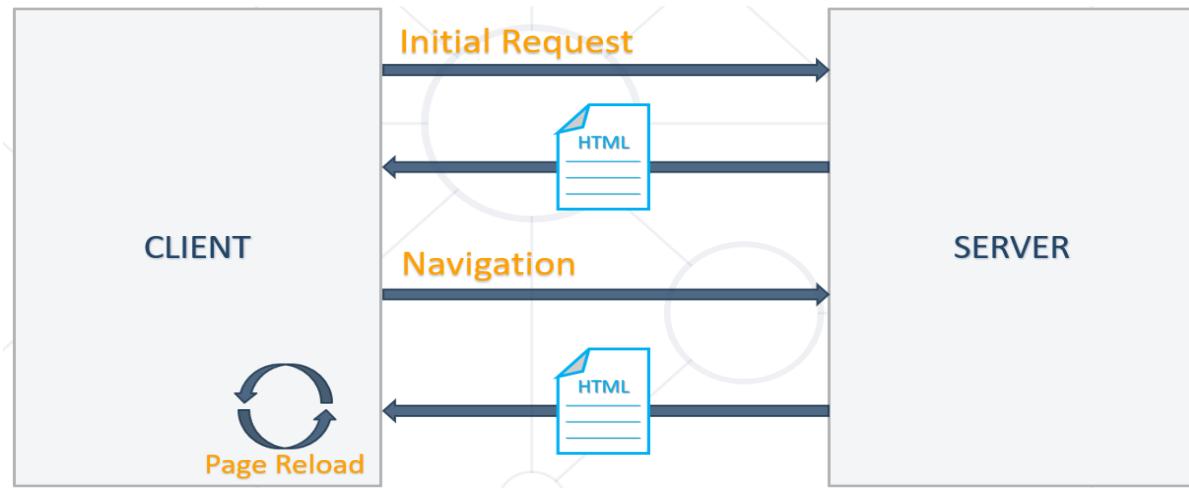
- A **website** is a collection of interlinked **web pages**
  - It hosts content, that is **primarily** meant to be **consumed**
- A **web application** is a **software**, accessible from a web browser
  - They are **interactive** and have rich **functionality**



#### Multi Page Applications

- **Reloads the entire page** – localSession и localStorage пази някакви данни
- **Displays the new page** when a user interacts with the web app
- When a data is exchanged, a **new page** is **requested** from the server to display in the web browser

#### Multi Page Application Lifecycle



#### Multi Page Pros and Cons

##### ■ Pros

- Performs well on the **search engine**
- Provides a **visual map** of the web app to the user

##### ■ Cons

- Comparatively **complex development**
- Coupled backend and frontend

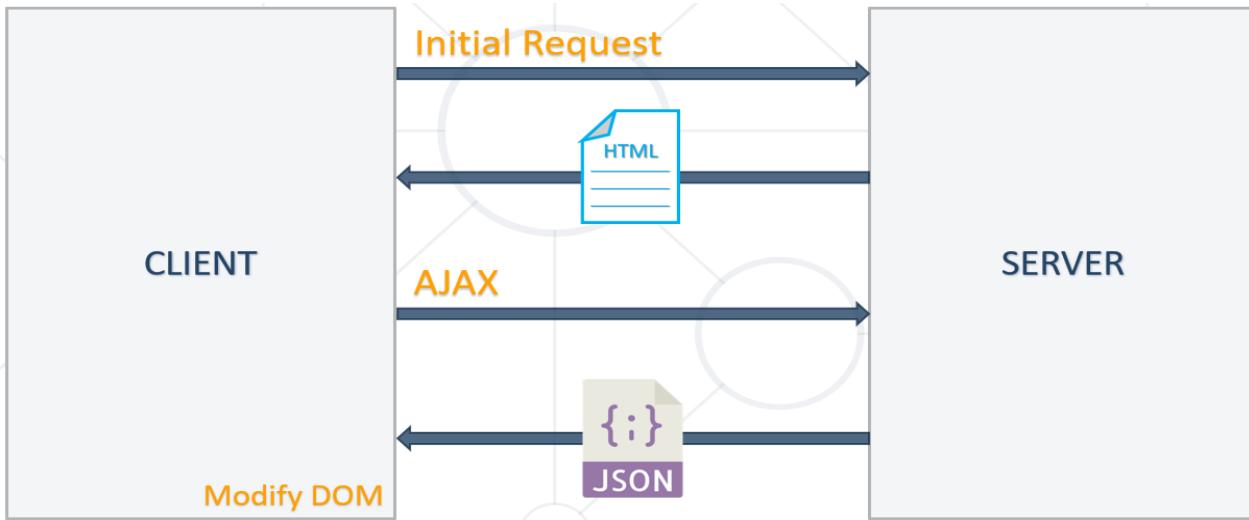
#### Single Page Applications

Да си заредим нашето приложение и то да си работи през цялото време, и то само от една единствена страница!

Да имаме споделена логика и състояния измежду всичките изгледи на тази страница, която да е споделена през цялото време. Глобални променливи, които могат да бъдат достъпни от всяка една точка от нашите скриптове.

- A next evolution from multi-page website
- Web apps that load a **single HTML file**
- SPAs use **AJAX** and **HTML5** to create fluid and responsive Web apps
- **No constant page reloads**

#### SPA Lifecycle



#### SPA Pros and Cons

- **Pros**
  - Load all scripts **only once**
  - **Maintain state** across multiple pages
  - Browser **history** can be used
  - Better **UX**
- **Cons**
  - Perform poor on the search engine
  - Server-side rendering helps
  - Provide **single sharing link**
  - Less secure

## 24.2. JavaScript Modules

### Modules

- A set of **related functionality**
  - Resolve naming collisions
  - Expose only public behavior
  - Not populate the global scope
- **Loaded** by setting the **type attribute** of a script:

```
<script src="app.js" type="module"></script>
```

При използване на модули, функциите са scope-нати в самите модули, и нямат директен глобален достъп през **window**. Трябва ръчно да ги закачим за **window**.

```
import { showHome } from "./home.js";
import { showDetails } from "./details.js";
import { showCreate } from "./create.js";
```

```
window.showHome = showHome;
window.showDetails = showDetails;
window.showCreate = showCreate;
```

- Note: Browsers **will not load** modules from the **file system** – you **must** use a local server
  - **lite-server** or similar

Code can be split and reused with **modules** – всеки файл = модул се грижи сам за себе си. Според зависи какво прави/върши всеки модул, използваме отделни модули или комбинация от различни модули - с import/export.

```

<html lang="en">
<head>
 <title>SPA Demo</title>
 <script src="src/app.js" type="module"></script>
</head>
<body>
 <nav>
 <button id="homeBtn">Home</button>
 <button id="catalogBtn">Catalog</button>
 <button id="aboutBtn">About us</button>
 </nav>

 <main>
 <h1>SPA Demo</h1>
 </main>

</body>
</html>

const main = document.querySelector('main');
document.querySelector('nav').addEventListener('click', onNavigate);

function onNavigate(event) {
 if (event.target.tagName == 'BUTTON') {
 switch (event.target.id) {
 case 'homeBtn': {
 main.innerHTML = `<h1>Home page</h1><p>Home page content</p>` //нов ред в
HTML – по-често използваме <p>some text here</p> отколкото

 break;
 }

 case 'catalogBtn': {
 main.innerHTML = `<h1>Catalog</h1>Product 1Product
2Product 3`;
 break;
 }

 case 'aboutBtn': {
 main.innerHTML = `<h1>About us</h1><p>About us content</p>`;
 break;
 }
 }
 }
}

```

```

 main.innerHTML = `<h1>About is</h1><p>Contact information <p>Phone: + 1-555-
7985</p> </p>`;
 break;
 }
}
}
}
}

```

ES6 Export Syntax - от един файл в друг

- **export** → expose public API

```

export function updateScoreboard(newResult) { ... }
export const homeTeam = 'Tigers';

```

- You can **export multiple** members

```
export { addResult, homeTeam as host };
```

- **Default exports** can be imported without a name

```
export default function addResult(newResult) { ... }
```

ES6 Import Syntax – от един файл в друг

- **import** → load **entire module** (all exports)

```

import * as scoreboard from './scoreboard.js';
scoreboard.updateScore(); // call from module

```

- Import **specific members** by name

```

import {addResult, homeTeam} from './scoreboard.js';
addResult(); // call directly by name

```

- Import **default export** by specifying alias

```

import addResult from './scoreboard.js'; // няма къдрави скоби тук при default export!!!
addResult(); // call directly by name

```

Legacy Approach – IIFE Modules

- **IIFE modules** were essential for larger projects
- They hide the unnecessary and expose only needed behavior/objects to the global scope

```

(function (scope) {
 const selector = 'loading';
 const loadingElement = document.querySelector(selector);
 const show = () => loadingElement.style.display = '';
 const hide = () => ladingElement.style.display = 'none';
 // Only this is visible to the global scope
 scope.loading = { show, hide };
})(window);

```

## Module Best Practices

- **Split code** in modules by related functionality
  - Aim for **high cohesion**
- Only export what is **necessary** for consumers
- Prefer **named exports** over defaults
- **Do not** perform operations on export

## 24.3. SPA Approaches

### SPA Implementation Requirements

- The application has **multiple views**
- All views **share** a common **state**
- **Modular code** is used
- The page is **not reloaded** when changing views
- Content is **loaded via AJAX**
- New **content (HTML elements)** is **created** by JavaScript

### Feasibility Disclaimer

- Contemporary single-page applications employ concepts like **templates** and **routing**
  - Usually with a **front-end framework**
- These topics will be covered in **upcoming lessons**
- The following approach aims to **demonstrate** the **basic principles** and is applicable only for **small apps**
  - Consider it for **educational purposes** only!

### Initial Static Content

- The **HTML template** holds all views as **hidden sections**
- Modules are responsible for **populating** and **displaying** views
- Sections can be controlled by **reference** or by **visibility** – **пазим референции на дом елементи и закачаме/откачаме от DOM дървото елементи.**
- **Dynamic content** is loaded via AJAX calls

Modular Code – всеки модул си има дадена функции/вид действие, което импортираме или експортраме

- Each **view** is controlled by its **own module**
  - Contains code for **fetching** and **displaying** related content
- A single script serves as the **application entry point**
  - **Imports** and **initializes** the rest of the modules
  - Holds and manages **shared (global) state**

## app.js

```
import HomePage from './home.js';
import catalogPage from './catalog.js';
// Load and initialize all modules
```

### Capturing Navigation

- **Anchor tags** instruct the browser to **navigate** to a new page
  - This will **restart** our application
- A **click handler** can be used to prevent this:

```
const navLink = document.getElementById('navLink');
navLink.addEventListener('click', e => {
 e.preventDefault();
 // Load new content and switch the view
});
```

- **View changes** can then be **triggered** from <a> elements

### Loading and Displaying Content

- Use the **Fetch API** to bring **new content** from the server
- **Modify** or **create** new HTML elements to **display content**

```
async function getArticles() {
 const response = await fetch(apiUrl);
 const articles = await response.json();
 articles.forEach(displayArticle);
}

function displayArticle(article) {
 // Modify DOM tree
}
```

### Group DOM Changes

- Manipulating the DOM tree is a **performance-intensive** process – на браузъра му се налага да преизчисли всички позиции на всички елементи наново
- When **multiple elements** must be created and populated, place them in a **DocumentFragment**:

```
const fragment = document.createDocumentFragment();
// Create and populate new elements
fragment.appendChild(/* element reference */);
document.body.appendChild(fragment); // Add to body
```

## 24.4. Проста примерна задача с ауторизация

```
▽ DEMO
 ▷ server
 ▷ src
 JS app.js
 JS catalog.js
 JS dom.js
 JS home.js
 JS login.js
 JS register.js
 ▷ demo_SPA_easy.txt
 ▷ index.html
```

### Index.html

```
<head>
 <title>SPA Demo</title>
 <script src="src/app.js" type="module"></script>
 <style>
 #sections {
 display: none;
 }
 label {
 display: block;
 }
 #guestNav, #userNav {
 display: none;
 }
 </style>
</head>

<body>
 <nav>
 <!-- <button id="homeBtn">Home</button>
 <button id="catalogBtn">Catalog</button>
 <button id="aboutBtn">About us</button> -->
 Home
 Catalog
 About us
 <div id="guestNav">
 Login
 Register
 </div>

 <div id="userNav">
 Logout
 </div>
```

```
Test without preventDefault() //линка не води към
никъде = страницата не се презарежда
</nav>

<main>
</main>

<div id="sections">
 <section id='homeSection'>
 <h1>Home Page</h1>
 <p>Welcome to our site!</p>
 </section>

 <section id='catalogSection'>
 <h1>Catalog</h1>

 Product 1
 Product 2
 Product 3

 </section>

 <section id="loginSection">
 <h1>Login</h1>
 <form>
 <label>Email: <input name="email" type="text" placeholder="Enter your email
address"> <!--задължително име трябва да имат, иначе formData няма да ги хване -->
 <label>Password: <input name="password" type="password" placeholder="Enter
your password"> </label>
 <input type="submit" value="login">
 </form>
 </section>

 <section id="registerSection">
 <h1>Register</h1>
 <form>
 <label>Email: <input name="email" type="text" placeholder="Enter your email
address"> <!--задължително име трябва да имат, иначе formData няма да ги хване -->
 <label>Password: <input name="password" type="password" placeholder="Enter
your password"> </label>
 <label>Repeat: <input name="repass" type="password" placeholder="Repeat your
password"> </label>
 <input type="submit" value="register">
 </form>
 </section>

 <section id='aboutSection'>
 <h1>About Us</h1>
 <p>Contact information</p>
 <p>Phone: +1-555-7985</p>
 </section>
</div>
```

```

 </section>
 </div>
</body>

dom.js
const main = document.querySelector('main');

export function showSection(section) {
 main.replaceChildren(section);
}

export function e(type, attributes, ...content) {
 const result = document.createElement(type);

 for (let [attr, value] of Object.entries(attributes || {})) {
 if (attr.substring(0, 2) == 'on') {
 result.addEventListener(attr.substring(2).toLocaleLowerCase(), value);
 } else {
 result[attr] = value;
 }
 }

 content = content.reduce((a, c) => a.concat(Array.isArray(c) ? c : [c]), []);
 //flatmap

 content.forEach(e => {
 if (typeof e == 'string' || typeof e == 'number') {
 const node = document.createTextNode(e);
 result.appendChild(node);
 } else {
 result.appendChild(e);
 }
 });
}

return result;
}

```

```

app.js
import { showCatalogPage } from "./catalog.js";
import { showHomePage, showAboutPage } from "./home.js";
import { showLoginPage } from "./login.js";
import { showRegisterPage } from "./register.js";

// const main = document.querySelector('main');
document.getElementById('logoutBtn').addEventListener('click', onLogout);
document.querySelector('nav').addEventListener('click', onNavigate);

//start application in home view
showHomePage();

```

```

const sections = {
 'homeBtn': showHomePage,
 'catalogBtn': showCatalogPage,
 'aboutBtn': showAboutPage,
 'loginBtn': showLoginPage,
 'registerBtn': showRegisterPage
}

updateUserNav();

function onNavigate(event) {
 console.log("Inside onNavigate");

 if (event.target.tagName == 'A') { //anchor //BUTTON - button
 const view = sections[event.target.id];

 console.log(view);

 if (typeof view == 'function') {
 event.preventDefault(); //не отивай към друга html страница
 view();
 }
 }
}

export function updateUserNav() {
 const userData = JSON.parse(sessionStorage.getItem('userData'));
 if (userData != null) {
 document.getElementById('userNav').style.display = 'inline-block';
 document.getElementById('guestNav').style.display = 'none';
 } else {
 document.getElementById('userNav').style.display = 'none';
 document.getElementById('guestNav').style.display = 'inline-block';
 }
}

async function onLogout(event) {
 event.stopImmediatePropagation(); //друг event listener (onNavigate) да не се изпълнява
 //върху същия anchor/button
 console.log("Inside onLogout");
 const {token} = JSON.parse(sessionStorage.getItem('userData'));

 await fetch('http://localhost:3030/users/logout', {
 headers: {
 'X-Authorization': token
 }
 });

 sessionStorage.removeItem('userData');
}

```

```
 updateUserNav();
 showHomePage();
 }

home.js
import { showSection } from "./dom.js";

const homeSection = document.getElementById('homeSection');
homeSection.remove(); //откачаме от DOM дървото - по-добър вариант от скриването

const aboutSection = document.getElementById('aboutSection');
aboutSection.remove(); //откачаме от DOM дървото - по-добър вариант от скриването
```

```
export function showHomePage() {
 showSection(homeSection);
}
```

```
export function showAboutPage() {
 showSection(aboutSection);
}
```

catalog.js

```
import { showSection, e } from "./dom.js";

const catalogSection = document.getElementById('catalogSection');
catalogSection.remove(); //откачаме от DOM дървото - по-добър вариант от скриването
const ul = catalogSection.querySelector('ul');
```

```
export function showCatalogPage() {
 showSection(catalogSection);
```

```
 loadMovies();
}
```

```
async function loadMovies() {
 ul.replaceChildren(e('p', {}, 'Loading...'));

 const options = {method: 'get', headers: {}};
 const userData = JSON.parse(sessionStorage.getItem('userData'));
 if (userData != null) {
 options.headers['X-Authorization'] = userData.token;
 }
 const response = await fetch('http://localhost:3030/data/movies', options);

 //В случай, че сървърът е рестартирал
 if (response.status == 403) {
 sessionStorage.removeItem('userData');
 updateUserNav();
```

```

 showLoginPage();
 }

 const movies = await response.json();
 ul.replaceChildren(...movies.map(createMovieCard));
}

function createMovieCard(movie) {
 return e('li', {}, movie.title);
}

```

```

login.js
import { updateUserNav } from "./app.js";
import { showSection } from "./dom.js";
import { showHomePage } from "./home.js";

const loginSection = document.getElementById('loginSection');
loginSection.remove(); //откачаме от DOM дървото - по-добър вариант от скриването
const form = loginSection.querySelector('form'); //формуларът от нашата секция
form.addEventListener('submit', onSubmit);
//кодът по-горе се изпълнява само веднъж при стартиране на браузъра, и няма как да се изпълни
//два пъти

export function showLoginPage() {
 showSection(loginSection);
}

async function onSubmit(event) {
 event.preventDefault();
 const formData = new FormData(form);

 const email = formData.get('email').trim();
 const password = formData.get('password').trim();

 try {
 const response = await fetch('http://localhost:3030/users/login', {
 method: 'post',
 headers: {
 'Content-Type': 'application/json'
 },
 body: JSON.stringify({ email, password })
 });

 if (response.ok != true) {
 const error = await response.json();
 throw new Error(error.message);
 }
 }

 const data = await response.json();

```

```

 const userData = {
 username: data.username,
 id: data._id,
 token: data.accessToken
 }
 sessionStorage.setItem('userData', JSON.stringify(userData));

 updateUserNav();
 showHomePage();
 } catch (err) {
 alert(err.message);
 }
}

```

register.js

```

import { updateUserNav } from "./app.js";
import { showSection } from "./dom.js";
import { showHomePage } from "./home.js";

const registerSection = document.getElementById('registerSection');
registerSection.remove(); //откачаме от DOM дървото - по-добър вариант от скридането
const form = registerSection.querySelector('form'); //формулярът от нашата секция
form.addEventListener('submit', onSubmit);
//кодът по-горе се изпълнява само веднъж при стартиране на браузъра, и няма как да се изпълни
//два пъти

export function showRegisterPage() {
 showSection(registerSection);
}

async function onSubmit(event) {
 event.preventDefault();
 const formData = new FormData(form);

 const email = formData.get('email').trim();
 const password = formData.get('password').trim();
 const repass = formData.get('repass').trim();

 if (password != repass) {
 alert('Passwords do not match');
 return;
 }

 try {
 const response = await fetch('http://localhost:3030/users/register', {
 method: 'post',
 headers: {
 'Content-Type': 'application/json'
 },

```

```

 body: JSON.stringify({ email, password })
 });

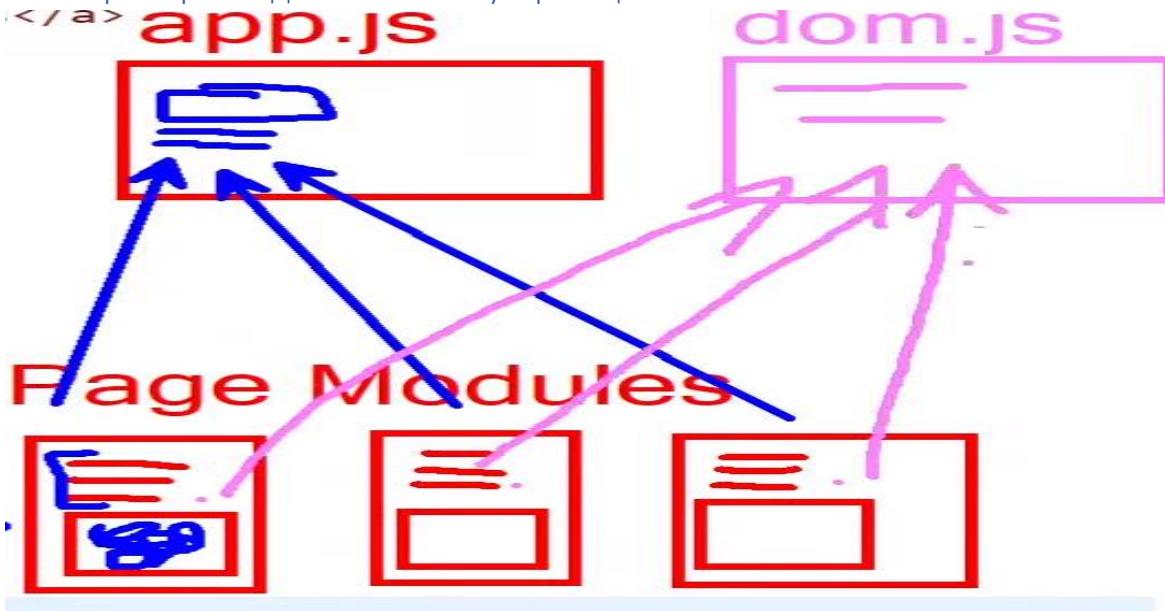
 if (response.ok != true) {
 const error = await response.json();
 throw new Error(error.message);
 }

 const data = await response.json();
 const userData = {
 username: data.username,
 id: data._id,
 token: data.accessToken
 }
 sessionStorage.setItem('userData', JSON.stringify(userData));

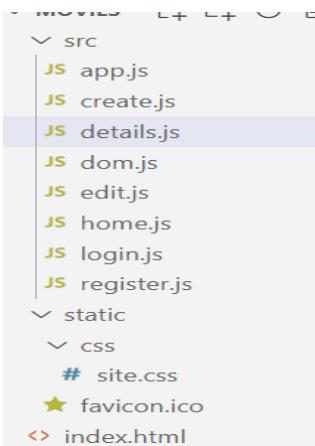
 updateUserNav();
 showHomePage();
} catch (err) {
 alert(err.message);
}
}
}

```

#### 24.5. Примерна задача с Likes и ауторизация



DDD – Domein Driver Development – папките и файловете в тях се кръщават спрямо областта на действие



```
app.js
import { showHome } from "./home.js";
import { showLogin } from "./login.js";
import { showRegister } from "./register.js";

//create placeholder modules for every view
const nav = document.querySelector("nav");

document.getElementById('logoutBtn').addEventListener('click', onLogout);
nav.addEventListener('click', (event) => {
 if (event.target.tagName == 'A') { //anchor or HTML-a
 const view = views[event.target.id];
 if (typeof view == 'function') {
 event.preventDefault();
 view();
 }
 }
});
updateNav();
//start application in home view (catalog)
showHome();

//configure and test navigation
const views = {
 'homeLink': showHome,
 'loginLink': showLogin,
 'registerLink': showRegister,
};

export function updateNav() {
 const userData = JSON.parse(sessionStorage.getItem('userData'));
 if (userData != null) {
 nav.querySelector('#welcomeMsg').textContent = `Welcome, ${userData.email}`;
 [...nav.querySelectorAll('.user')].forEach(d => d.style.display = 'block');
 [...nav.querySelectorAll('.guest')].forEach(d => d.style.display = 'none');
 } else {

```

```

 [...nav.querySelectorAll('.user')].forEach(d => d.style.display = 'none');
 [...nav.querySelectorAll('.guest')].forEach(d => d.style.display = 'block');
 }
}

async function onLogout(event) {
 event.preventDefault();
 event.stopImmediatePropagation();
 const {token} = JSON.parse(sessionStorage.getItem('userData'));

 //get request
 await fetch('http://localhost:3030/users/logout', {
 headers: {
 'X-Authorization': token
 }
 });

 sessionStorage.removeItem('userData');
 updateNav();
 showLogin();
}

//implement modules
///-- create async functions for requests
///-- implement DOM logic

// Order of views:
//x catalog (home view)
//x login
//x logout
// - register - едно if допълнително има спрямо login
// - create film
//x details //x likes
// - edit film
// - delete film

```

details.js

```

//initialization
// - find relevant section

import { showView, e } from "./dom.js";

// - detach section from DOM
const section = document.getElementById('movie-details');
section.remove();

//display logic

```

```

export function showDetails(movieId) {
 showView(section);
 getMovie(movieId);
}

async function getMovie(id) {
 section.replaceChildren(e('p', {}, "Loading..."));
 const requests = [fetch('http://localhost:3030/data/movies/' + id),
 fetch(`http://localhost:3030/data/likes?where=movieId%3D${id}%22&distinct=_ownerId&cou
nt`)];

 const userData = JSON.parse(sessionStorage.getItem('userData'));
 if (userData != null) {
 requests.push(fetch(`http://localhost:3030/data/likes?where=movieId%3D${id}%22%20a
nd%20_ownerId%3D${userData.id}%22`));
 }

 const [movieRes, likesRes, hasLikedRes] = await Promise.all(requests);

 const [movieData, likes, hasLiked] = await Promise.all([
 movieRes.json(),
 likesRes.json(),
 hasLikedRes && hasLikedRes.json() //това се ползва навсякъде, и в React също
]);

 section.replaceChildren(createDetails(movieData, likes, hasLiked));
}

function createDetails(movie, likes, hasLiked) {
 console.log(hasLiked);
 const controls = e('div', { className: 'col-md-4 text-center' },
 e('h3', { className: 'my-3' }, 'Movie Description'),
 e('p', {}, movie.description)
);

 const userData = JSON.parse(sessionStorage.getItem('userData'));

 // debugger; //стартира в който и да е браузър под дебъг режим
 if (userData != null) {
 if (userData.id == movie._ownerId) {
 controls.appendChild(e('a', { className: 'btn btn-danger', href: '#' },
 'Delete'));
 controls.appendChild(e('a', { className: 'btn btn-warning', href: '#' },
 'Edit'));
 } else {
 if (hasLiked.length > 0) {
 controls.appendChild(e('a', { className: 'btn btn-primary', href: '#',
 onClick: onUnlike }, 'Unlike'));
 } else {

```

```

 controls.appendChild(e('a', { className: 'btn btn-primary', href: '#',
onClick: onLike }, 'Like')));
 }
}

controls.appendChild(e('span', { className: 'enrolled-span' }, `Liked ${likes}`));

const element = e('div', { className: 'container' },
 e('div', { className: 'row bg-light text-dark' },
 e('h1', {}, `Movie title: ${movie.title}`),
 e('div', { className: 'col-md-8' },
 e('img', { className: 'img-thumbnail', src: movie.img, alt: 'Movie' })
),
 controls
)
);
return element;

async function onLike() {
 await fetch('http://localhost:3030/data/likes', {
 method: 'post',
 headers: {
 'Content-type': 'application/json',
 'X-Authorization': userData.token
 },
 body: JSON.stringify({
 movieId: movie._id
 })
 });
 showDetails(movie._id);
}

async function onUnlike() {
 const likeId = hasLiked[0]._id; //благодарение на closure знаем от кой потребител е
лайка

 await fetch('http://localhost:3030/data/likes/' + likeId, {
 method: 'delete',
 headers: {
 'X-Authorization': userData.token
 },
 });
 showDetails(movie._id);
}
}

```

```
home.js
//initialization
// - find relevant section

import { e, showView } from "./dom.js";
import { showCreate } from "./create.js";
import { showDetails } from "./details.js";

let moviesCache = null;

// - detach section from DOM
const section = document.getElementById('home-page');
const catalog = section.querySelector('.card-deck.d-flex.justify-content-center');
section.querySelector('#createLink').addEventListener('click', (event) => {
 event.preventDefault();
 showCreate();
});

catalog.addEventListener('click', (event) => {
 event.preventDefault();
 let target = event.target;
 if (target.tagName == 'BUTTON') {
 target = target.parentElement;
 }

 if (target.tagName == 'A') {
 const id = target.dataset.id;
 showDetails(id);
 }
});
section.remove();

//display logic
export function showHome() {
 showView(section);

 getMovies();
}

async function getMovies() {
 catalog.replaceChildren(e('p', {}, 'Loading...'));
 if (moviesCache == null) {
 console.log('first load');
 const res = await fetch('http://localhost:3030/data/movies');
 const data = await res.json();
 moviesCache = data;
 }

 catalog.replaceChildren(...moviesCache.map(createMovieCard));
}
```

```

}

function createMovieCard(movie) {
 const element = e('div', { classname: 'card mb-4' });
 element.innerHTML =

<div class="card-body">
 <h4 class="card-title">${movie.title}</h4>
</div>
<div class="card-footer">
 <a data-id=${movie._id} href="#">
 <button type="button" class="btn btn-info">Details</button>

</div>';
 return element;
}
// window.getMovies = getMovies;

```

dom.js

```

const main = document.querySelector('main');

export function showView(section) {
 main.replaceChildren(section);
}

//function that creates elements
export function e(type, attributes, ...content) {
 const result = document.createElement(type);

 for (let [attr, value] of Object.entries(attributes || {})) {
 if (attr.substring(0, 2) == 'on') {
 result.addEventListener(attr.substring(2).toLocaleLowerCase(), value);
 } else {
 result[attr] = value;
 }
 }

 content = content.reduce((a, c) => a.concat(Array.isArray(c) ? c : [c]), []);
 content.forEach(e => {
 if (typeof e == 'string' || typeof e == 'number') {
 const node = document.createTextNode(e);
 result.appendChild(node);
 } else {
 result.appendChild(e);
 }
 });
}
```

```
 return result;
 }


```

create.js

```
//initialization
// - find relevant section

import { showView } from "./dom.js";

// - detach section from DOM
const section = document.getElementById('add-movie');
section.remove();

//display logic
export function showCreate() {
 showView(section);
}


```

edit.js

```
//initialization
// - find relevant section

import { showView } from "./dom.js";

// - detach section from DOM
const section = document.getElementById('edit-movie');
section.remove();

//display logic
export function showEdit() {
 showView(section);
}


```

login.js

```
import { updateNav } from "./app.js";
import { showView } from "./dom.js";
import { showHome } from "./home.js";

//initialization
// - find relevant section
// - detach section from DOM
const section = document.getElementById('form-login');
const form = section.querySelector('form');
form.addEventListener('submit', onLogin);
section.remove();
```

```

//display logic
export function showLogin() {
 showView(section);
}

async function onLogin(event) {
 event.preventDefault();
 const formData = new FormData(form);

 const email = formData.get('email').trim();
 const password = formData.get('password').trim();

 try {
 const response = await fetch('http://localhost:3030/users/login', {
 method: 'post',
 headers: {
 'Content-type': 'application/json'
 },
 body: JSON.stringify({email, password})
 });

 if (response.ok == false) {
 const error = await response.json();
 throw new Error(error.message);
 }

 const data = await response.json();
 sessionStorage.setItem('userData', JSON.stringify({
 email: data.email,
 id: data._id,
 token: data.accessToken
 }));
 form.reset(); //reset-ваме формуляра

 updateNav();
 showHome();
 } catch (err){
 alert(err.message);
 }
}

```

`register.js – това не е разписано - едно if допълнително има спрямо login`

```

//initialization
// - find relevant section

import { showView } from "./dom.js";

// - detach section from DOM

```

```
const section = document.getElementById('form-sign-up');
const form = section.querySelector('form');
form.addEventListener('submit', onRegister);
section.remove();

//display logic
export function showRegister() {
 showView(section);
}

async function onRegister(event) {
 event.preventDefault();
 const formData = new FormData(form);

 const email = formData.get('email').trim();
 const password = formData.get('password').trim();
 const repeatPassword = formData.get('repeatPassword').trim();

 try {
 const response = await fetch('http://localhost:3030/users/register', {
 method: 'post',
 headers: {
 'Content-type': 'application/json'
 },
 body: JSON.stringify({email, password})
 });

 if (response.ok == false) {
 const error = await response.json();
 throw new Error(error.message);
 }

 const data = await response.json();
 sessionStorage.setItem('userData', JSON.stringify({
 email: data.email,
 id: data._id,
 token: data.accessToken
 }));
 form.reset(); //reset-ваме формуляра

 updateNav();
 showHome();
 } catch (err){
 alert(err.message);
 }
}
```

## 25. Architecture and Testing

### 25.1. Separating Concerns(разделяне на отговорности) - Writing Easy to Maintain Code

#### Drawbacks of Mixed Concerns

- **Multiple concerns** – parts of the application perform actions on **various domains** (e.g., DB calls, business logic, UI)
- This leads to **high coupling**:
  - **Low abstraction** level limits the size of the application
  - It's **difficult to change** one module without affecting the rest
  - **Code steps are repeated** out of necessity
  - It's **impractical to reuse** a module in another applications
  - The developer must be **aware of all specifics** of every module

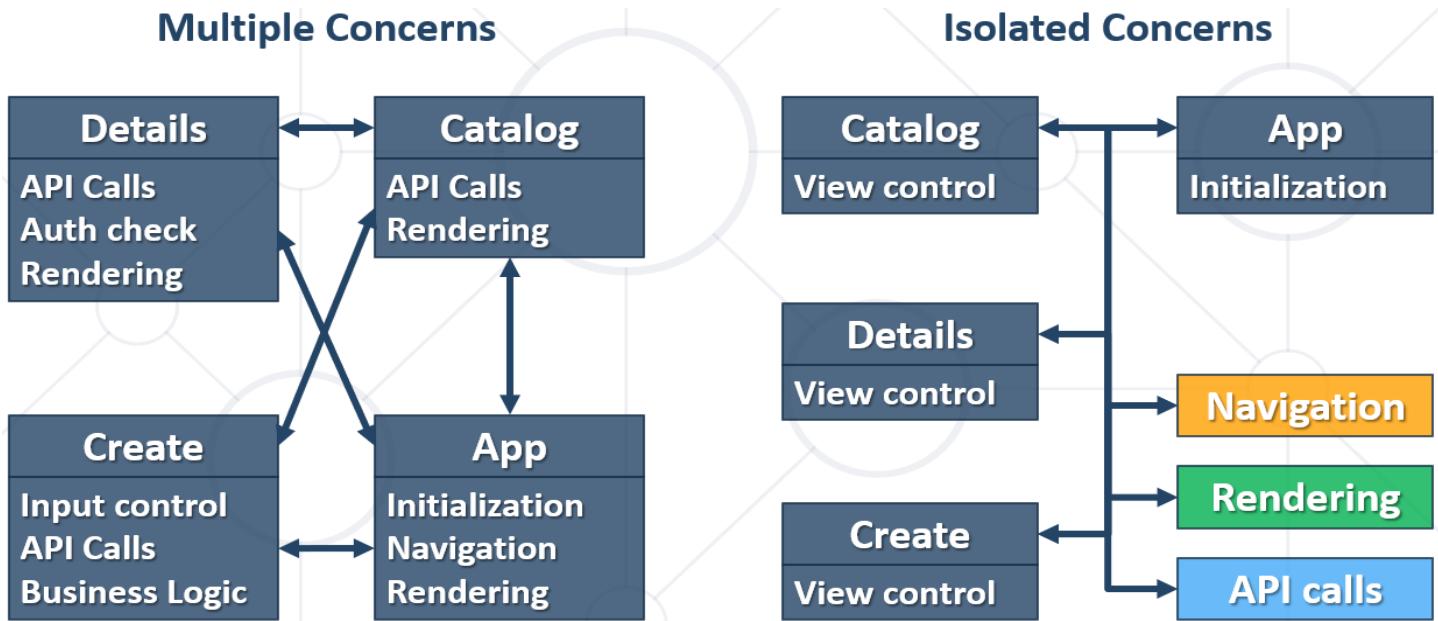
#### Goal of Separation of Concerns

- Limit a unit of code (function, module) to a **single domain**
  - E.g., a method that **only visualizes** (renders) data on screen
- Implementation is **abstract from details**
  - E.g., the rendering function **does not concern** itself with the source of the data
- The developer **doesn't need to know** how a module operates internally in order to use it
- **Code reuse** is a secondary effect – **easier reasoning** is primary

#### Extracting Functionality into Modules

- Common steps:
  - **Extract actions** over different domains in their own functions
  - Identify **similar actions** across different parts of the application
  - **Increase abstraction** of the extracted functions, so that they can be **used in more places** with minimal changes
  - Move functions from a single domain to a **separate module**
- **Don't overdo abstraction!** A good rule of thumb – increase abstraction **when** you need to **refactor** the code

## Isolated Modules



## Example Isolated Modules

- **Backend API** – specific to the used service
- **Request logic** – specific to the application business logic
- **Data manipulation** – specific to the application business logic
- **UI display and control**
- **Utility functions**

## 25.2. Application Testing

### Types of Tests

- **Unit tests** – cover separated functionality
  - E.g., test the **result of a function** with different input
- **Integration tests** – cover the communication inside and between entire modules
  - E.g., test if data coming from a **remote request** is correctly interpreted by the **business logic**
- **End-to-end (Functional) tests** – cover **all steps** that occur when the **user** performs an **action**, from the UI, to the DB, and back

### Unit Tests Usage

- **Unit tests** are used to verify that a **piece of code** (function, class, etc.) operates correctly
- The tested code does **not** involve **external dependencies** (application state, other modules, external systems)
- They are fast to **write** and fast to **execute**
- Usually **created by the developer**, who is aware of the code specifics (**white-box** testing)
- Common tools include **Mocha**, **Chai**, **QUnit**, **Jasmine**, etc.

### Integration Tests Usage

- **Integration tests** are used to check the communication between multiple code elements (functions, classes, entire modules)

- They often require the **inclusion of external dependencies** (other application modules, databases, remote resources)
- Relatively **complex to create** (due to the external dependencies)
- Can be delegated to a **separate team**, not involved in the writing of the code (**black-box testing**)
- Common tools include **Sinon, JMock, Mockito**, etc.

### End-to-End (Functional) Tests Usage

- **Functional tests** are used to run through the **entire application**, in a real environment
- Usually involves the whole **technological stack** (REST services, database operations, authentication, etc.)
- Depending on the expected outcome and tools used, their **complexity** is comparable to **integration tests**
- Mostly the concern of specialized **QA automation engineers**
- Common tools include **Selenium, Puppeteer, Cypress**, etc.

## 25.3. Testing with Playwright

### What is Playwright?

- A **complete suite for testing web applications** in a real environment – the **web browser**
  - Our application is executed inside a "headless" browser
  - User **input is simulated**, and the result is **monitored**
- Compatible with **Chromium, Firefox** and **WebKit**
- Available in **JavaScript, TypeScript, Python, C#** and **Java**
- Home page: <https://playwright.dev/>

### Installation and Environment

- Install via **NPM** with **Chromium support**:

**npm init -y** – да има отделна папка само за тестовете и без други задачи

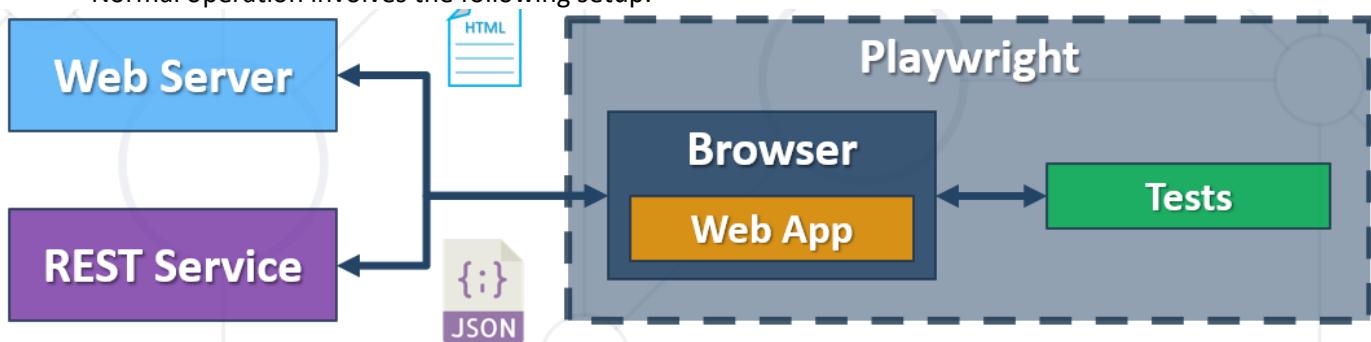
**npm install --save-dev playwright-chromium**

**npm install -g mocha**

**npm install chai**

Ако дадем команда **npm install --save-dev playwright**, то ще инсталира за трите браузъра Chromium, Firefox and WebKit

- **Note:** this will download **browser binaries (~200 MB)**
- Normal operation involves the following setup:



- Create **test.js** and enter the following code:

```
const { chromium } = require('playwright-chromium');
(async () => {
 const browser = await chromium.launch(); //зареди браузърът за тестове
 const browser = await chromium.launch({headless: false, slowMo: 2000}); //браузъре, бъди
видим и чакай по 2 секунди между всяко действие.
 const page = await browser.newPage(); //дай му нов таб/празна страница
 await page.goto('https://google.com/'); //тази страница да зареди определения сайт
 await page.screenshot({ path: `example.png` }); //направи screenshot на дадената страница и
го запиши в example.png
 await browser.close();
})();
```

- Execute via Node.js:

```
node test.js
```

### Project Setup

- Combine with a **test-running framework** (e.g., Mocha and Chai)

```
const { chromium } = require('playwright-chromium');
const { expect } = require('chai');
let browser, page; // Declare reusable variables
describe('E2E tests', async function () {
 this.timeout(5000); //единственият начин да имаме this е като имаме обикновена функция

 before(async () => { browser = await chromium.launch(); });
 after(async () => { await browser.close(); });
 beforeEach(async () => { page = await browser.newPage(); }); //преди всеки тест
 afterEach(async () => { await page.close(); }); //след всеки тест

 it('works', async () => {
 await (new Promise(response => setTimeout(response, 2000))); //тестваме дали работи и
при асинхронна среда
 expect(1).to.equal(1);
 })
});
```

- **Note:** make sure both the **REST service** and **web server** are running **before** executing tests

Example: Loading Static Page

*Direct navigation – same as entering the URL in the address-bar*

```
const { chromium } = require('playwright-chromium');
const { expect } = require('chai');
let browser, page; // Declare reusable variables
describe('E2E tests', async function () {
 this.timeout(5000);
```

```

before(async () => { browser = await chromium.launch(); });
after(async () => { await browser.close(); });
beforeEach(async () => { page = await browser.newPage(); });
afterEach(async () => { await page.close(); });

it('initial load', async () => {
 await page.goto('http://localhost:5500');
 await page.screenshot({path: 'page.png'});
})
});

```

*Visiting via clicking on links (<a>-tags)*

```

await page.click('a[href="/register"]');
await page.waitForNavigation();
await page.waitForLoadState();
// Perform operations on new page

```

Example: Finding Elements

*CSS Selectors:*

```

await page.click('button'); // Basic selector
await page.click('article:has(div.promo)'); // Content-based

```

*Find element by text content*:- търсим елементи по съдържанието на текста на DOM елемента

Примерно, думичката на бутона, който потребителят вижда

```

// Case-insensitive, partial matches
await page.click('text=Log in');
// Case-sensitive, full match only
await page.click('text="Log in"');

```

*via \$\$eval()*

```

//селектираме каквото си искаме и го преобразуваме в масив
const rows = await page.$$eval('tr', rows => rows.map(r => r.textContent.trim()));
//$$eval() е добре/задължително да го използваме в контролирана среда
console.log(rows);
expect(rows[1]).to.contains('Harry Potter');
expect(rows[1]).to.contains('Rowling');
expect(rows[2]).to.contains('C# Fundamentals');
expect(rows[2]).to.contains('Nakov');

```

```
eval(new String('2 + 2'));
```

```
eval('2 + 2');
```

```

var expression = new String('2 + 2');

eval(expression.toString());

```

Example: Verifying Content

Obtain text content:

```
const content = await page.textContent(selector);
page.textContent('text=Harry Potter'); - намира първия срещнат DOM елемент с такъв текст

it('initial load', async () => {
 await page.goto('http://localhost:5500');

 await page.waitForSelector('.accordion'); //Чакаме докато браузърът не зареди нещо с
 // клас accordion, await спира изпълнението на кода надолу докато не се появи исканото

 const content = await page.textContent('#main');
 expect(content).to.contains('Scalable Vector Graphics');
 expect(content).to.contains('Open standard');
 expect(content).to.contains('Unix');
 expect(content).to.contains('ALGOL');
})
```

Attribute value:

```
const val = await page.getAttribute(selector, attrName);
```

Checkbox state:

```
const checked = await page.isChecked(selector);
```

Visibility:

```
const visible = await page.isVisible(selector);
```

```
const DEBUG = true;
```

```
it.only('more buttons test', async () => { //като сложим .only се изпълнява само този тест
 await page.goto('http://localhost:5500');
 await page.waitForSelector('.accordion');

 await page.click('text=More');//case insensitive, ще намери първия такъв елемент

 // await page.waitForRequest(); // това няма смисъл, трябва да изчакаме отговора от
 // сървъра
 await page.waitForResponse(/articles\/details/i);

 await page.waitForSelector('.accordion p'); //може да не е нужен този ред
 const visible = await page.isVisible('.accordion p');

 expect(visible).to.be.true;
})
```

## Example: Form Input

*Text input:*

```
await page.fill(selector, 'Peter'); // Text
await page.fill(selector, '2020-02-02'); // Date
await page.fill('text=First Name', 'Peter'); // Via label
```

*Checkboxes and radio buttons:*

```
await page.check(selector);
await page.uncheck(selector);
```

*Select options (single and multiple values):*

```
await page.selectOption(selector, 'blue');
await page.selectOption(selector, ['red', 'green', 'blue']);
```

*Selector chaining*

Advanced usage: <https://playwright.dev/docs/selectors>

## Example: Request Handling

*Submit form and wait for response:*

Тук НЕ интересува резултата от изпращане на заявката, и затова използваме `waitForResponse`

```
const [response] = await Promise.all([
 page.waitForResponse('**/api/data'),
 page.click('input[type="submit"]'),
]);
```

*Request matching can be done with predicate:*

```
page.waitForResponse(
 response => response.url().includes(token));
```

*Obtain request body (to validate sent values):*

```
const postData = JSON.parse(response.request().postData());
```

*Submit form and wait for request:*

Тук НЕ интересува резултата от изпращане на заявката, а само съдържанието на Заявката, и затова използваме `waitForRequest`

```
const DEBUG = true;
it.only('can create a book', async () => {
 await page.goto('http://localhost:5500');
```

```

// В playwright > търси едно ниво надолу спрямо предния селектор, докато >> търси в
// дълбочина навсякъде в елемента
await page.fill('form#createForm > input[name="title"]', 'Title A');
await page.fill('form#createForm > input[name="author"]', 'Author B');

const [reqquest] = await Promise.all([
 page.waitForRequest(request => request.method = 'POST'), // чакаме за каквато и
да е post заявка
 page.click('form#createForm > text=Submit')
]);

const data = JSON.parse(reqquest.postData());
console.log(data);
expect(data.title).to.equal('Title A');
expect(data.author).to.equal('Author B');
});

```

### Example: Response Mocking

*Setup request interception can return mock data:*

- Note: this must be configured before the form is submitted

Регистрира слушател .route

```

await page.route('**/api/data', route => route.fulfill({
 status: 200,
 body: testData,
}));

```

The screenshot shows a browser's developer tools Network tab. A single request is listed with the following details:

- Status Code: 200 OK
- Remote Address: [::1]:3030
- Referrer Policy: strict-origin-when-cross-ori
- Response Headers (expanded):
  - HTTP/1.1 200 OK
  - Access-Control-Allow-Origin: \*
  - Content-Type: application/json
  - Date: Mon, 03 Jan 2022 20:46:24 GMT

```

// Две звездички означава хващай всички hostove
await page.route('**/jsonstore/collections/books*', (rout) => {
 rout.fulfill({
 status: 200,
 headers: {
 'Access-Control-Allow-Origin': '*',
 'Content-Type': 'application/json'
 },
 body: JSON.stringify(mockData)
 });
});

```

*Abort requests (to prevent external calls or resource loading):*

АбORTиране зареждане на външни ресурси обикновено

```
await page.route('**/*.{png,jpg,jpeg}',
 route => route.abort());
```

С **Ctrl + C** в терминала спираме изпълнението, в случая изпълнението на теста

25.4. Същата проста примерна задача с ауторизация РЕФАКТУРИРАНЕ - вдигане на абстракцията и прилагане на dependency injection

**Всяка задача от JS Application да се рънва в своята си папка!!!**

`await`-а чака грешката. Грешката/промиса се прехвърля по веригата. Ако не `await`-нем, при наличие на грешка, то грешката отлиза/изчезва. Така са навързани грешките в JS.

Модул = файл

Оставили сме всички модули да подготвят сами себе си

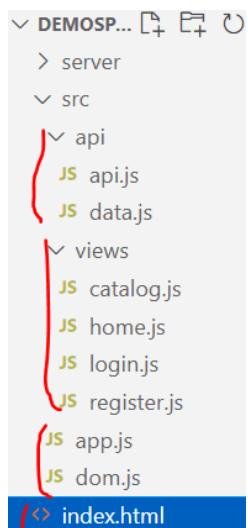
При отворени **ВСИЧКИ** файлове в таба, като ги местим в друга папка, Visual Studio Code автоматично ни преименува пътя на импортите!!!

`./` - текуща папка

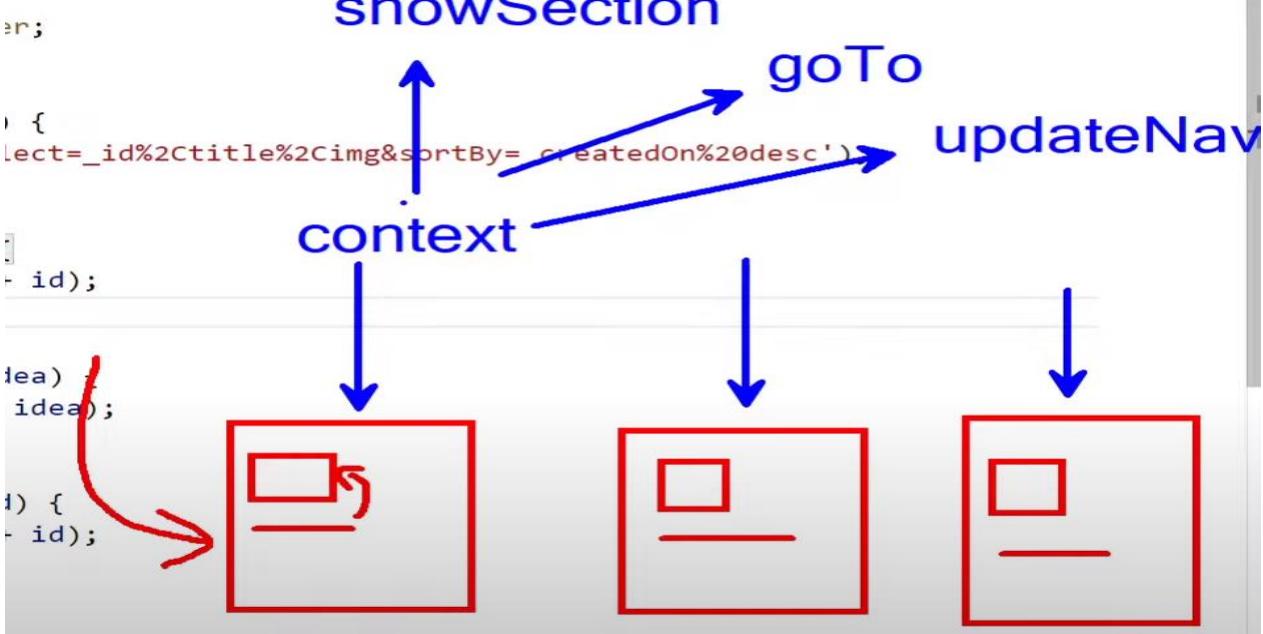
`../` - една папка по-горе

`../../` - две папки нагоре

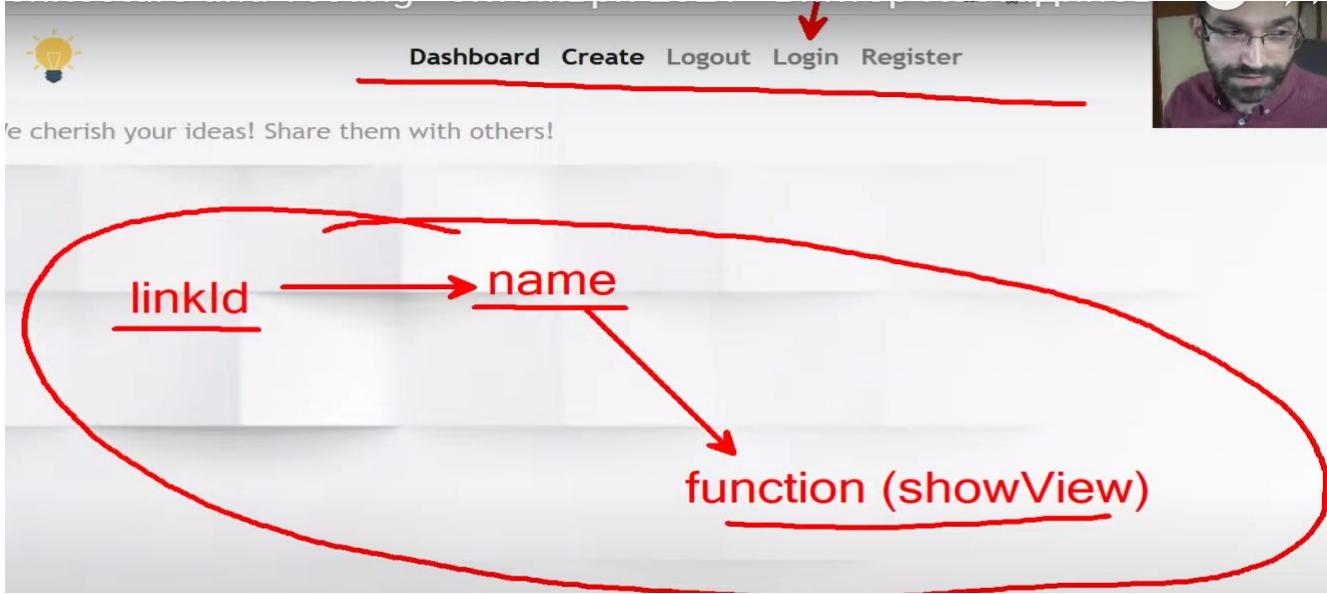
`/` - абсолютна директория/папка – спрямо старта на нашето приложение



Идеята за инжектиране на context / ctx



Линк ID на кое view name съответства, и view name на коя функция се изпълнява (контекста, да ѝ инженктнем нужното dependency)



index.html

```
<!DOCTYPE html>
<html lang="en">

<head>
 <title>SPA Demo</title>
```

```
<script src="src/app.js" type="module"></script>
<style>
 #sections {
 display: none;
 }
 label {
 display: block;
 }
 #guestNav, #userNav {
 display: none;
 }
</style>
</head>

<body>
 <nav>
 <!-- <button id="homeBtn">Home</button>
 <button id="catalogBtn">Catalog</button>
 <button id="aboutBtn">About us</button> -->
 Home
 Catalog
 About us
 <div id="guestNav">
 Login
 Register
 </div>

 <div id="userNav">
 Logout
 </div>

 Test without preventDefault()
 </nav>

 <main>
 </main>

 <div id="sections">
 <section id='homeSection'>
 <h1>Home Page</h1>
 <p>Welcome to our site!</p>
 </section>

 <section id='catalogSection'>
 <h1>Catalog</h1>

 Product 1
 Product 2
 Product 3

 </section>
 </div>

```

```

 </section>

 <section id="loginSection">
 <h1>Login</h1>
 <form>
 <label>Email: <input name="email" type="text" placeholder="Enter your email address"> <!--задължително име да имат, иначе formData няма да ги хване -->
 <label>Password: <input name="password" type="password" placeholder="Enter your password"> </label>
 <input type="submit" value="login">
 </form>
 </section>

 <section id="registerSection">
 <h1>Register</h1>
 <form>
 <label>Email: <input name="email" type="text" placeholder="Enter your email address"> <!--задължително име да имат, иначе formData няма да ги хване -->
 <label>Password: <input name="password" type="password" placeholder="Enter your password"> </label>
 <label>Repeat: <input name="repass" type="password" placeholder="Repeat your password"> </label>
 <input type="submit" value="register">
 </form>
 </section>

 <section id='aboutSection'>
 <h1>About Us</h1>
 <p>Contact information</p>
 <p>Phone: +1-555-7985</p>
 </section>
 </div>

</body>
</html>

```

dom.js

```

const main = document.querySelector('main');

export function showSection(section) {
 main.replaceChildren(section);
 // document.querySelectorAll('section').forEach(s => s.style.display = 'none');
 // document.getElementById(sectionId).style.display = '';
}

export function e(type, attributes, ...content) {
 const result = document.createElement(type);

 for (let [attr, value] of Object.entries(attributes || {})) {

```

```

 if (attr.substring(0, 2) == 'on') {
 result.addEventListener(attr.substring(2).toLocaleLowerCase(), value);
 } else {
 result[attr] = value;
 }
 }

content = content.reduce((a, c) => a.concat(Array.isArray(c) ? c : [c]), []);
content.forEach(e => {
 if (typeof e == 'string' || typeof e == 'number') {
 const node = document.createTextNode(e);
 result.appendChild(node);
 } else {
 result.appendChild(e);
 }
});
return result;
}

```

```

app.js
import { logout } from "./api/data.js";
import { showCatalogPage } from "./views/catalog.js";
import { showHomePage, showAboutPage } from "./views/home.js";
import { showLoginPage } from "./views/login.js";
import { showRegisterPage } from "./views/register.js";
import {showSection} from "./dom.js"

// const main = document.querySelector('main');
document.getElementById('logoutBtn').addEventListener('click', onLogout);
document.querySelector('nav').addEventListener('click', onNavigate);

// скритите видове страници, div-ове
const views = {
 'home': showHomePage,
 'catalog':showCatalogPage,
 'details':showDetailsPage,
 'about':showAboutPage,
 'login': showLoginPage,
 'register': showRegisterPage
};

// линкове от nav менюто
const links = {
 'homeBtn': 'home',
 'catalogBtn':'catalog',
 'aboutBtn': 'about',
 'loginBtn': 'login',

```

```

 'registerBtn': 'register'
};

//context = dependency injection
const ctx = {
 updateUserNav,
 goTo,//ето я функцията/контекста
 showSection,//ето я функцията/контекста
};

debugger; //start debugging at this point of the code

//start application in home view
updateUserNav();
goTo('home');

function onNavigate(event) {
 console.log("Inside onNavigate");

 if (event.target.tagName == 'A') {//anchor //BUTTON - button
 const name = links[event.target.id];
 if (name) {
 event.preventDefault(); //не отивай към друга html страница
 goTo(name);
 }
 }
}

function goTo(name, ...params) {//тези параметри не са за goTo(name). Те са за другите
 //функции, които викат goTo. Примерно ще бъде id на детайлите/DOM/HTML елементите.
 const view = views[name];
 if (typeof view == 'function') {
 view(ctx, ...params);
 }
}

export function updateUserNav() {
 const userData = JSON.parse(sessionStorage.getItem('userData'));
 if (userData != null) {
 document.getElementById('userNav').style.display = 'inline-block';
 document.getElementById('guestNav').style.display = 'none';
 } else {
 document.getElementById('userNav').style.display = 'none';
 document.getElementById('guestNav').style.display = 'inline-block';
 }
}

//или този код
if (userData != null) {
 [...nav.querySelectorAll('.user')].forEach(l => l.style.display = 'block');
 [...nav.querySelectorAll('.guest')].forEach(l => l.style.display = 'none');
}

```

```

} else {
 [...nav.querySelectorAll('.user')].forEach(l => l.style.display = 'none');
 [...nav.querySelectorAll('.guest')].forEach(l => l.style.display = 'block');
}

}

async function onLogout(event) {
 event.stopImmediatePropagation(); //друг event listener (onNavigate) да не се изпълнява
 //върху същия anchor/button
 console.log("Inside onLogout");

 //from ./api/data.js
 await logout();

 updateUserNav();
 goTo('home');
}

```

views/catalog.js

```

import { getAllMovies } from '../api/data.js';
import { e } from '../dom.js';

const catalogSection = document.getElementById('catalogSection');
catalogSection.remove(); //откачаме от DOM дървото - по-добър вариант от скридането
const ul = catalogSection.querySelector('ul');

export function showCatalogPage(ctx, id) { //параметрите на goTo функцията
 ctx.showSection(catalogSection);
 loadMovies();
}

async function loadMovies() {
 ul.replaceChildren(e('p', {}, 'Loading...'));
 const movies = await getAllMovies();
 ul.replaceChildren(...movies.map(createMovieCard));
}

function createMovieCard(movie) {
 return e('li', {}, movie.title);
}

```

views/details.js – пример за използване на параметър/Dom елемент на goTo функцията

```

import { e } from '../dom.js';

const section = document.getElementById('detailsPage');
section.remove();
section.addEventListener('click', onDetails);

```

```

let ctx = null; //за да използваме ctx както през injection-а чрез функцията
showDetailsPage(ctxTarget, id), така и директно в scope-а на текущия модул/файл.

export async function showDetailsPage(ctxTarget, id) { //параметрите на goTo функцията
 ctx = ctxTarget;
 ctx.showSection(section);
 loadIdea(id);
}

function onDetails(ev) {
 if(ev.target.tagName == 'A') {
 const id = ev.target.dataset.id;
 ev.preventDefault();
 ctx.goTo('details', id); //това пак вика функцията showDetailsPage(ctxTarget, id)
 }
}

async function loadIdea(id) {
 const idea = await getById(id);
 export async function getById(id) {
 return api.get('/data/ideas/' + id);
 }
 section.replaceChildren(createIdeaDiv(idea));
}

function createIdeaDiv(idea) {
 const fragment = document.createDocumentFragment();

 fragment.appendChild(e('img', {className: 'det-img', src: idea.img}));

 fragment.appendChild(e('div', {className: 'desc'},
 e('h2', {className: 'display-5'}, idea.title),
 e('p', {className: 'infoType'}, 'Description:'),
 e('p', {className: 'idea-description'}, idea.description)
));
 fragment.appendChild(e('div', {className: 'text-center'}),
 e('a', {className: 'btn detb', href: ""}, 'Delete')
);
}

 return fragment;
}

```

views/home.js

```

const homeSection = document.getElementById('homeSection');
homeSection.remove(); //откачаме от DOM дървото - по-добър вариант от скриването

```

```
const aboutSection = document.getElementById('aboutSection');
aboutSection.remove(); //откачаме от DOM дървото - по-добър вариант от скриването

export function showHomePage(ctx, id) { //параметрите на goTo функцията) {
 ctx.showSection(homeSection);
}

export function showAboutPage(ctx) {
 ctx.showSection(aboutSection);
}
```

views/login.js

```
import * as api from '../api/data.js';//една папка нагоре

const loginSection = document.getElementById('loginSection');
loginSection.remove(); //откачаме от DOM дървото - по-добър вариант от скриването
const form = loginSection.querySelector('form'); //формулярът от нашата секция
form.addEventListener('submit', onSubmit);
//кодът по-горе се изпълнява само веднъж при стартиране на браузъра, и няма как да се изпълни
//две пъти

let ctx = null;

export function showLoginPage(ctxTarget, id) { //параметрите на goTo функцията) {
 ctx = ctxTarget;
 ctx.showSection(loginSection);
}

//Имаме достъп отново през контекста да си извикаме updateUserNav и showHomePage
async function onSubmit(event) {
 event.preventDefault();
 const formData = new FormData(form);

 const email = formData.get('email').trim();
 const password = formData.get('password').trim();

 if (!email || !password) {
 return alert("All fields are required");
 }

 await api.login(email, password);
 form.reset();
 ctx.updateUserNav();
 ctx.goTo('home'); //може да няма инфраструктура готова, за да го приложим
}
```

```

views/register.js

import { register } from "../api/data.js"; //една папка нагоре

const registerSection = document.getElementById('registerSection');
registerSection.remove(); //откачаме от DOM дървото - по-добър вариант от скриването
const form = registerSection.querySelector('form'); //формулярът от нашата секция
form.addEventListener('submit', onSubmit);
//кодът по-горе се изпълнява само веднъж при стартиране на браузъра, и няма как да се изпълни
//два пъти

let ctx = null;

export function showRegisterPage(ctxTarget, id) { //параметрите на goTo функцията
 ctx = ctxTarget;
 ctx.showSection(registerSection);
}

async function onSubmit(event) {
 event.preventDefault();
 const formData = new FormData(form);

 const email = formData.get('email').trim();
 const password = formData.get('password').trim();
 const repass = formData.get('repass').trim();

 if (!email || !password) {
 return alert("All fields are required");
 }

 if (password != repass) {
 alert('Passwords do not match');
 return;
 }

 //from ./api/data.js
 await register(email, password);
 form.reset();
 ctx.updateUserNav();
 ctx.goTo('home');
}

```

api/api.js – тук за http заявките

```

const host = 'http://localhost:3030';

//функция на ниско ниво, извършва проверките
async function request(url, options) {
 try {

```

```

const response = await fetch(host + url, options);

if (response.ok != true) { //има network грешка
 if (response.status == 403) { //има проблем с token при аутентификация на user
 sessionStorage.removeItem('userData'); //изтриваме, за да можем да позволим
на потребителя да се логне наново
 }

 const error = await response.json(); //network-а връща грешка, която стои в body-
to
 throw new Error(error.message);
}

// try {
// const data = await response.json();
// return data;
// } catch (er) {
// return response;
// }

if (response.status == 204) { //в response няма данни /No content
 return response;
} else {
 return response.json(); //няма нужда от await в случая, връща Promise, който ще
се await-не впоследствие
}

} catch (err) {
 alert(err.message)
 throw err; //прехвърля грешката нататък, за да може всички да разберат какво се е
случило
}
}

//тази функция също е на ниско ниво
function createOptions(method = 'get', data) {
 const options = {
 method,
 headers: {}
 }

 if (data != undefined) {
 options.headers['Content-Type'] = 'application/json';
 options.body = JSON.stringify(data);
 }

 const userData = JSON.parse(sessionStorage.getItem('userData'));
 if (userData != null) {
 options.headers['X-Authorization'] = userData.token;
 }
}

```

```
 return options;
}

//функции на високо ниво
async function get(url) {
 return request(url, createOptions());
}

//функции на високо ниво
async function post(url, data) {
 return request(url, createOptions('post', data));
}

//функции на високо ниво
async function put(url, data) {
 return request(url, createOptions('put', data));
}

//функции на високо ниво
async function del(url) {
 return request(url, createOptions('delete'));
}

async function login(email, password) {
 const response = await request('/users/login', createOptions('post', {email, password}));
 const userData = {
 email: response.email,
 id: response._id,
 token: response.accessToken
 };
 sessionStorage.setItem('userData', JSON.stringify(userData));
}

async function register(email, password) {
 const response = await request('/users/register', createOptions('post', {email, password}));
 const userData = {
 email: response.email,
 id: response._id,
 token: response.accessToken
 };
 sessionStorage.setItem('userData', JSON.stringify(userData));
}

async function logout() {
 //результатата не ни интересува, но трябва да го изчакаме
 await request('/users/logout', createOptions());

 sessionStorage.removeItem('userData');
```

```
}
```

```
export {
 get,
 post,
 put,
 del,
 login,
 register,
 logout
}
```

```
api/data.js
```

```
import * as api from './api.js';

const endpoints = {
 movies: '/data/movies'
}

export async function getAllMovies() {
 return api.get(endpoints.movies);
}

export const login = api.login;
export const register = api.register;
export const logout = api.logout;

export async function getById(id) {
 return api.get('/data/ideas/' + id);
}

export async function createIdea(idea) {
 return api.post('/data/ideas', idea);
}

export async function deleteById(id) {
 return api.del('/data/ideas/' + id);
}
```

## 26. Client-Side Rendering – шаблонизация на елементи от потребителския интерфейс (Templating UI Elements)

### 26.1. UI Rendering

Rendering Concepts / Интерпретация/визуализиране на нещо на екрана

- **Rendering means to dynamically generate content**
  - As opposed to having **static** HTML files (такива, които съществуват като файл)

- Can be **parts** of a web page, or an **entire web application**
- Virtually **all contemporary sites** use dynamic generation
- Can be performed on the **server** and on the **client** (browser)

## Server-Side Rendering



## Client-Side Rendering

### Server-Side vs Client-Side

#### Server-Side

- User sends request
- Server **generates HTML**
- **HTML is sent** to the client
- Browser **interprets** HTML

#### Client-Side

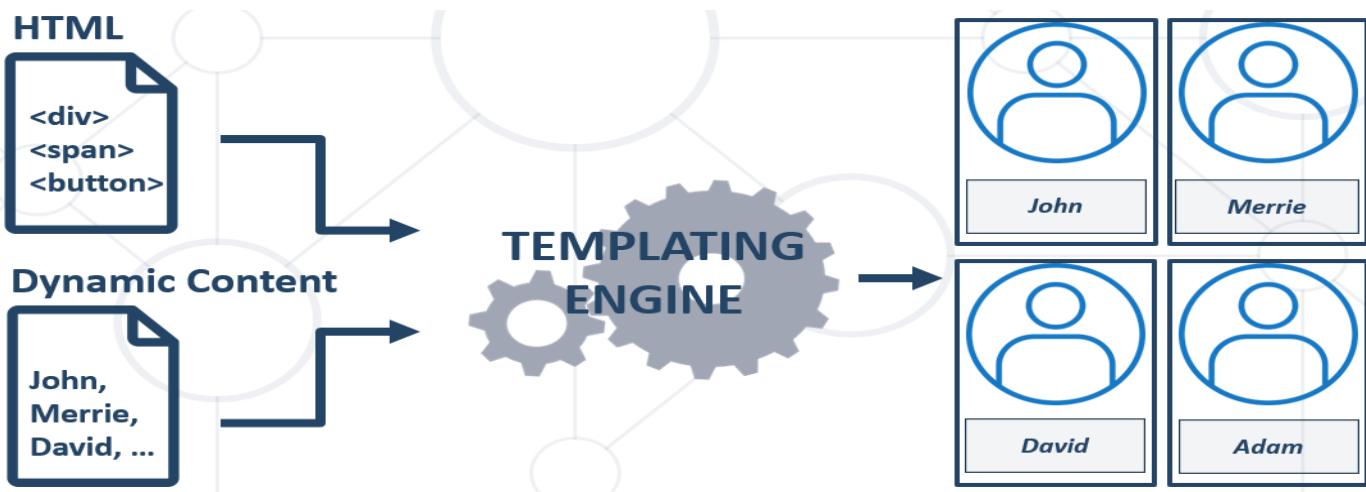
- User sends request
- CDN(Content delivery network) serves **files** and **JS** – например Live Server, да изпраща само данните
- JS **fetches** data
- JS **generates DOM elements**

### Pros and Cons of Client-Side Rendering

- **Benefits:**
  - The page is **never reloaded**, and interaction is **instant**
  - State and data can be **shared** across views
  - **Only** dynamic content needs to be **fetched** after start
- **Drawbacks:**
  - Longer **initial load** times
  - **Not SEO**(search engine optimization)-friendly
  - Poor **performance** with **slow client** machines

### What is Templating?

- Templates allow similar content to be **replicated** in a web page, **without repeating** the corresponding markup everywhere
- Размножаваме HTML елементите и в тях попълваме динамичното съдържание



### Templating Concepts

- On the **server**, templates are used to **generate HTML**
  - E.g., content from a **database** is inserted into **placeholders**
- On the **client**, templates are used to **create DOM elements**
  - The **template** defines the **structure** of a view
  - Content is **fetched** from a **REST service**
  - The structure is **recreated** and **populated** with the data
  - A **templating engine** is used to streamline the process

### Templating Benefits

- **Productivity** – avoid repeating markup(HTML)
- **Save bandwidth/трафик данни** – fetch just the dynamic content
- **Composability** – reuse elements on multiple pages
- **Separation of concerns / разделяне на отговорностите** – separate views from logic
- **Interactivity** – instant feedback to the user

### Templating Best Practices

- Templates should be as **simple** as possible
  - **Do not** write business logic in the templates
- Follow the principles of **functional programming**
  - Templates are basically **pure functions**

## 26.2. Custom Templates

### Creating a Simple Templating Engine

#### Project Requirements

- A templating engine **generally** allows:
  - Templates to be **defined** in files, **separate** from business logic
  - A **markup syntax** close to HTML to be used
  - Values to be inserted via **rendering context**
  - Templates to be **composed** to create **layouts**
- **Additional features** of some libraries:

- **Caching** of template results
- **Automating** diff-checking and **partial updates**

## Creating a Simple Templating Engine

### DEMOS

```

JS app.js
<> article.html
{} data.json
JS engine.js
<> index.html
site.css

```

### *index.html*

```

<!DOCTYPE html>
<html lang="en">

<head>
 <meta charset="UTF-8">
 <meta http-equiv="X-UA-Compatible" content="IE=edge">
 <meta name="viewport" content="width=device-width, initial-scale=1.0">
 <title>Templating Demo</title>
 <link rel="stylesheet" href="site.css">
 <script src=".//app.js" type="module"></script>
</head>

<body>
 <h1>Templating Demo</h1>
 <main>
 <article>
 <h3>First Article</h3>
 <div class="content-body">
 <p>
 Lorem ipsum dolor sit amet, consectetur adipisicing elit. Ab quidem
 accusantium voluptatum nulla adipisci ipsa voluptates assumenda asperiores? Doloribus
 aspernatur quis architecto ullam quae ipsa unde molestias aliquam soluta corporis.
 </p>
 </div>
 <footer>Author: John Smith</footer>
 </article>

 <article>
 <h3>First Article</h3>
 <div class="content-body">
 <p>
 Lorem ipsum dolor sit amet consectetur adipisicing elit. Dolorum harum a
 alias dolore error sit consequatur, architecto ipsum perferendis dolores expedita repellat
 </p>
 </div>
 </article>
 </main>
</body>

```

placeat. Animi modi ex molestias. Accusantium obcaecati, laboriosam illo eum officia sint maiores repellendus. Doloribus ex quisquam, earum mollitia vitae, ullam adipisci voluptates facilis nesciunt at similique molestias eum. Voluptatum molestiae debitis aperiam aliquam saepe laboriosam veniam, sed omnis amet! Inventore atque nisi aperiam tenetur quidem quia eveniet id voluptatibus velit sed eum laudantium quisquam natus, magnam aliquam error quo nemo reiciendis tempore, assumenda deleniti? Vel exercitationem expedita commodi officia corporis porro asperiores, magni vero amet. Optio, voluptatibus!

```
 </p>
 </div>
 <footer>Author: John Smith</footer>
</article>
</main>

</body>
</html>
```

data.json - данните

```
[

 {
 "title": "First article",
 "content": "Lorem ipsum dolor sit amet consectetur adipisicing elit. Dolorum harum a alias dolore error sit consequatur, architecto ipsum perferendis dolores expedita repellat placeat. Animi modi ex molestias. Accusantium obcaecati, laboriosam illo eum officia sint maiores repellendus. Doloribus ex quisquam, earum mollitia vitae, ullam adipisci voluptates facilis nesciunt at similique molestias eum. Voluptatum molestiae debitis aperiam aliquam saepe laboriosam veniam, sed omnis amet! Inventore atque nisi aperiam tenetur quidem quia eveniet id voluptatibus velit sed eum laudantium quisquam natus, magnam aliquam error quo nemo reiciendis tempore, assumenda deleniti? Vel exercitationem expedita commodi officia corporis porro asperiores, magni vero amet. Optio, voluptatibus!",
 "author": "John Smith"
 },
 {
 "title": "Second article",
 "content": "Lorem ipsum dolor sit amet consectetur adipisicing elit. Dolorum harum a alias dolore error sit consequatur, architecto ipsum perferendis dolores expedita repellat placeat. Animi modi ex molestias. Accusantium obcaecati, laboriosam illo eum officia sint maiores repellendus. Doloribus ex quisquam, earum mollitia vitae, ullam adipisci voluptates facilis nesciunt at similique molestias eum. Voluptatum molestiae debitis aperiam aliquam saepe laboriosam veniam, sed omnis amet! Inventore atque nisi aperiam tenetur quidem quia eveniet id voluptatibus velit sed eum laudantium quisquam natus, magnam aliquam error quo nemo reiciendis tempore, assumenda deleniti? Vel exercitationem expedita commodi officia corporis porro asperiores, magni vero amet. Optio, voluptatibus!",
 "author": "Peter Johnson"
 },
 {
 "title": "Third article",
 "content": "Lorem ipsum dolor sit amet consectetur adipisicing elit. Dolorum harum a alias dolore error sit consequatur, architecto ipsum perferendis dolores expedita repellat placeat. Animi modi ex molestias. Accusantium obcaecati, laboriosam illo eum officia sint
```

```

maiores repellendus. Doloribus ex quisquam, earum mollitia vitae, ullam adipisci voluptates
facilis nesciunt at similique molestias eum. Voluptatum molestiae debitis aperiam aliquam
saepe laboriosam veniam, sed omnis amet! Inventore atque nisi aperiam tenetur quidem quia
eveniet id voluptatibus velit sed eum laudantium quisquam natus, magnam aliquam error quo
nemo reiciendis tempore, assumenda deleniti? Vel exercitationem expedita commodi officia
corporis porro asperiores, magni vero amet. Optio, voluptatibus!",

 "author": "Marie Curie"
}

]

```

*article.html – шаблонът*

```

<article>
 <h3>{{title}}</h3>
 <div class="content-body">
 <p>{{content}}</p>
 </div>
 <footer>Author: {{author}}</footer>
 <div class="comments">
 <p>Some comments</p>
 </div>
</article>

```

*app.js*

```

import { renderTemplate } from "./engine.js";

async function start() {
 const data = await (await fetch('./data.json')).json(); //това го правихме на 2 реда, сега
 //правим на един ред
 const template = await (await fetch('./article.html')).text(); //HTML не е json

 const main = document.querySelector('main');

 // const result = renderTemplate(template, data[0]);

 //Един и същи template го подаваме на различни артикли
 main.innerHTML = data.map(artl => renderTemplate(template, artl)).join("");
}

start();

```

*engine.js*

```

export function renderTemplate(templateAsString, data) {
 //по принцип с regex не се parse-ва html текст
 const pattern = /{{(.+?)}}/gm;
 return templateAsString.replace(pattern, (match, propName) => {
 return data[propName];
 // console.log(propName, data[propName]);
 });
}

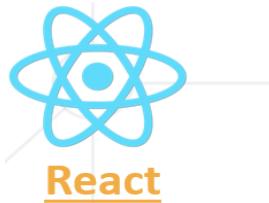
```

```
});
}
```

## 26.3. Templating Engines

### Overview of Popular JS Libraries

- Frameworks – дава структура за цялостно приложение:



- Standalone Packages – решава само проблема с rendering на templating-a:



Handlebars



lit-html



Web Components

lit-components

## 26.4. Lit-html External Templating Library

### Using **lit-html** to Generate Content from Templates

#### What is lit-html?

- **lit-html** is an efficient, expressive **templating library**

```
let sayHello = (name) => html`<h1>Hello ${name}</h1>`;//тагнат литерал / темплейт литерал
//html е функция от lit-html библиотеката, която ще премине през интерполирания стринг и ще
генерира html елементи спрямо подадения текст
render(sayHello('World'), document.body);
```

- Part of the Polymer Project
- Allows **rendering** and **partial updating** of templates
- Uses **standard JavaScript and HTML syntax**
- Can be **customized** and extended
- **Compatible** with all major browsers

#### Getting Started

- Installation via **npm** package – това е правилния начин за изпита в СофтУни:

**npm init -y**

**npm install lit-html**

```
import { html, render } from './node_modules/lit-html/lit-html.js';
```

- Installation only for stylization (style highlighting) when writing code in the tilda placeholder ` `:



- Direct **import** from online **CDN** (no installation):

```
import {html, render} from 'https://unpkg.com/lit-html?module'; //библиотеката ще се свали на браузъра, и браузърът си я вика
```

- Online **live editors**:

- [CodeSandbox](#)
- [JSBin](#)
- [StackBlitz](#)

## Usage

- To use lit-html, **import** it as a module:

```
<script type="module">
 import {html, render} from './node_modules/lit-html/lit-html.js'; //Path to main file
(use live-server to start)

...
</script>
```

```
let sayHello = (name) => html`<h1>Hello ${name}</h1>`;
render(sayHello('World'), document.body);
```

## Rendering a Template

- lit-html has two main APIs:
  - The **html template tag** used to write templates.
  - The **render()** function used to render a template to a DOM container

```
app.js file
import {html, render} from 'https://unpkg.com/lit-html?module';

const articleTemplate = (dataEl) =>
 html`<article>
 <h3>${dataEl.title}</h3>
 <div class="content-body">
 <p>${dataEl.content}</p>
 </div>
 <footer>Author: ${dataEl.author}</footer>
 <div class="comments">
```

```

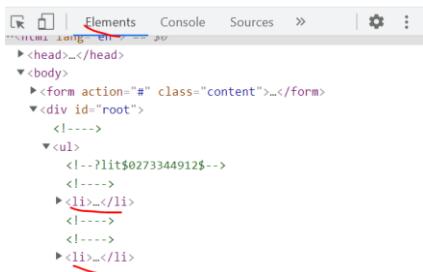
 <p>Some comments</p>
 </div>
</article>`;
```

start();

```

async function start() {
 const data = await (await fetch('./data.json')).json(); //това го правихме на 2 реда, сега
 //го правим на един ред
 const main = document.querySelector('main');
 const renderBtn = document.getElementById('renderBtn');
 renderBtn.addEventListener('click', onRender);
```

function onRender(ev) {
 data[0].author += '1'; //тук променяме името на автора, и при натискане наново на
 //бутон renderBtn от html-а, то render функцията на lit-html променя спрямо предходното
 //извикване/натискане на бутона само промяната!!! Всички dom операции не се случват, освен
 //необходимата, по която има промяна. В различните браузъри, като дадем на elements светва в
 //жълт цвят само промяната или в лилав цвят – зависи от браузъра.



```

const result = data.map(articleTemplate);

 render(result, main);
}
}
```

*data.json*

```
[
{
 "title": "First article",
 "content": "Lorem ipsum dolor sit amet, consectetur adipisicing elit. Ab quidem
 accusantium voluptatum nulla adipisci ipsa voluptates assumenda asperiores? Doloribus
 aspernatur quis architecto ullam quae ipsa unde molestias aliquam soluta corporis.",
 "author": "John Smith",
 "color": "red"
},
{
 "title": "Second article",
 "content": "Lorem ipsum dolor sit amet, consectetur adipisicing elit. Ab quidem
 accusantium voluptatum nulla adipisci ipsa voluptates assumenda asperiores? Doloribus
 aspernatur quis architecto ullam quae ipsa unde molestias aliquam soluta corporis.",
 "author": "Peter Johnson",
}
```

```

 "color": "green"
 },
{
 "title": "Third article",
 "content": "Lorem ipsum dolor sit amet, consectetur adipisicing elit. Ab quidem
accusantium voluptatum nulla adipisci ipsa voluptates assumenda asperiores? Doloribus
aspernatur quis architecto ullam quae ipsa unde molestias aliquam soluta corporis.",
 "author": "Marie Curie",
 "color": "blue"
}
]

```

## Tag Functions / Tagged Templates

- A tagged template is a **function call** that uses a **template literal** from which to get its arguments

```
// Tag Function Call
greet`I'm ${name}. I'm ${age} years old.`
```

- Create a greet function and just log the arguments:

```
function greet() {
 console.log(arguments[0]); // array
 console.log(arguments[1]); // name
 console.log(arguments[2]); // age
}
```

## Attribute Binding – задаване стойност на атрибути на HTML елементи

- In addition to using expressions in the text content of a node, you can bind them to a node's attribute and property values, too:

```

▼<article>
 ▷ <h3>...</h3>
 ▷ <div class="red">...</div>
 ▷ <footer>...</footer>
 ▷ <div class="comments">...</div>
</article>

const articleTemplate = (dataEl) =>
 html`<article>
 <h3>${dataEl.title}</h3>
 <div class="${dataEl.color}">
 <div class=${dataEl.color}> парсъра го слага в кавички автоматично
 <p>${dataEl.content}</p>
 </div>
 <footer>Author: ${dataEl.author}</footer>
 <div class="comments">
 <p>Some comments</p>
 </div>
 </article>`;

```

- Use the **?** prefix for a **boolean** attribute binding: - булев атрибут

*data.json*

```
{
 "title": "Second article",
 "content": "Lorem ipsum dolor sit amet, consectetur adipisicing elit. Ab quidem
accusantium voluptatum nulla adipisci ipsa voluptates assumenda asperiores? Doloribus
aspernatur quis architecto ullam quae ipsa unde molestias aliquam soluta corporis.",
 "author": "Peter Johnson",
 "color": "green",
 "disabled": true
},
```

*app.js*

```
const articleTemplate = (dataEl) =>
 html`

<input type="text" ?disabled=${dataEl.disabled}>
 <h3>${dataEl.title}</h3>
 <div class="${dataEl.color}">
 <p>${dataEl.content}</p>
 </div>
 <footer>Author: ${dataEl.author}</footer>
 <div class="comments">
 <p>Some comments</p>
 </div>
 </article>`;


```

▼ **<article>**

|   **<input type="text" disabled>**

## Property Binding

Нещата, които се интерпретират от DOM, а не от HTML, то ги пишем с точка отпред

- You can also bind to a **node's JavaScript properties** using the **. prefix** and the property name:

```
const articleTemplate = (dataEl) =>
 html`

<input type="text" ?disabled=${dataEl.disabled} .value=${dataEl.color}> //чрез lit-html
 така задаваме стойност на полето
 <h3>${dataEl.title}</h3>
 <div class="${dataEl.color}">
 <p>${dataEl.content}</p>
 </div>
 <footer>Author: ${dataEl.author}</footer>
 <div class="comments">
 <textarea .value=${"Hello there"}></textarea> //чрез lit-html така задаваме стойност
 на полето
 </article>`;


```

```
</div>
</article>`;
```

- You can use property bindings to pass complex data down the tree to subcomponents – когато сме си направили собствен компонент

```
const myTemplate = (data) => html`<my-list .listItems=${data.items}></my-list>`;
```

## Handling Events

- Templates can also include declarative event listeners
- An event listener looks like an attribute binding, but with the **prefix @** followed by an **event name**:

```
const appRootTemplate = (ctx) => html`
 <div>
 <h1 @click=${ctx.handleClick}>${ctx.title}</h1>
 </div>
`
```

## Conditional Statements

- lit-html has **no built-in control-flow** constructs. Instead you use normal JavaScript expressions and statements:

```
html`
 ${user.isLoggedIn}
 ? html`Welcome ${user.name}`
 : html`Please log in`
`
`;
```

## List Rendering

- To render lists, you can use **Array.map** to transform a list of data into a list of templates:

```
html`

 ${items.map((item) => html`${item}`)}

`;
```

```
app.js
const articleTemplate = (onSubmit, dataEl) =>
 html`

<article>

 <input type="text" ?disabled=${dataEl.disabled} .value=${dataEl.color}> //добавяме така

DOM пропърти в lit-html

 <h3>${dataEl.title}</h3>

 <div class="${dataEl.color}">

 <p>${dataEl.content}</p>

 </div>

 <footer>Author: ${dataEl.author}</footer>
```

```

<div class="comments">
 <form @submit=${onSubmit}>
 <textarea name="comment"></textarea>
 <button>Submit comment</button>
 </form>

 ${dataEl.comments.map(commentTemplate)}

</div>
</article>`;

const commentTemplate = (comment) => html`${comment.content}`;

data.json
[
 {
 "title": "First article",
 "content": "Lorem ipsum dolor sit amet, consectetur adipisicing elit. Ab quidem accusantium voluptatum nulla adipisci ipsa voluptates assumenda asperiores? Doloribus aspernatur quis architecto ullam quae ipsa unde molestias aliquam soluta corporis.",
 "author": "John Smith",
 "color": "red",
 "comments": []
 },

```

## Directives: classes and classMap

- The **classMap directive** lets you set a **group of classes attributes** based on an object:

```

import { classMap } from './node_modules/lit-html/directives/class-map.js';
import { classMap } from 'https://unpkg.com/lit-html/directives/class-map.js?module';

```

### Пример 1:

```

const itemTemplate = (item) => {
 const classes = { selected: item.selected };
 return html`<div class="menu-item" ${classMap(classes)}>Classy text</div>`;
}

```

### Пример 2:

```

const indexTemplate = (quizId, i, isCurrent, isAnswered) => {
 const className = {
 'q-index': true, //this class will be present in the attributes of the tag <a>
 'q-current': isCurrent,
 'q-answered': isAnswered
 };
 return html``;
}

```

## Directives: styles and styleMap

- You can use the **styleMap directive** to set **inline styles** on an element in the template:

```
import { styleMap } from './node_modules/lit-html/directives/style-map.js';
import { styleMap } from 'https://unpkg.com/lit-html/directives/style-map.js?module';
```

```
const styles = {
 color: myTextColor,
 backgroundColor: highlight ? myHighlightColor : myBackgroundColor
};
```

**styleMap** е функция, която приема асоциативен масив (обект демек) от стилове  
html`<div style=\${styleMap(styles)}>Hi there!</div>`;

### Option 1 - Декларативен стил - Directives: style-map.js

Декларативен стил – ето ти данните, ти се визуализирай на базата на тяхното съдържание. Вместо да казваме на html елементи ти вече си видим, то казваме на темплейта твоите данни се промениха. Това е принципа, на който работи React, Vue, Angular

Императивен стил – при тези и тези обстоятелства, направи това и това.

app.js

```
import { contacts } from './contacts.js';
import {html, render} from './node_modules/lit-html/lit-html.js';
import {styleMap} from './node_modules/lit-html/directives/style-map.js'; //css променя

const contactTemplate = (data, onDetails) => html`

<div>
 <i class="far fa-user-circle gravatar"></i>
 </div>
 <div class="info">
 <h2>Name: ${data.name}</h2>
 <button class="detailsBtn" @click={() => onDetails(data)}> ${data.details ? 'Hide' : 'Details'}
 </button>
 <div class="details" id="${data.id}" style=${styleMap({display: data.details ? 'block' : 'none'})}> <!– В случая може и да не използваме styleMap. styleMap се използва за още по-сложни css атрибути-->
 <p>Phone number: ${data.phoneNumber}</p>
 <p>Email: ${data.email}</p>
 </div>
 </div>
</div>
`;

start();


```

```

function start() {
 const container = document.getElementById('contacts');

 onRender();

 function onDetails(contact) {
 contact.details = !contact.details; // добавяме ново свойство details на всеки щракнат
обект – ако е true element.style.display = ‘block’, иначе element.style.display = ‘none’
 onRender();
 }

 function onRender() {
 render(html`#${contacts.map(c => contactTemplate(c, onDetails))}`, container);
 }
}

```

При всяка една модификация викаме една и съща функция и казваме изгради елементите наново. Lit-html е достатъчно умен да изгради само промените.

Отпърваме се от големия проблем – управлението на състоянието на DOM-а. Не се налага експлицитно да ходим и да ръчкаме конкретни DOM елементи и да не знаем дали някой скрипт на страницата е ръчнал грешния DOM елемент.

#### *Option 2 - Вариант на предната задача с делегиране*

app1.js

```

import { contacts } from './contacts.js';
import {html, render} from './node_modules/lit-html/lit-html.js';

const contactTemplate = (data) => html`

Name: ${data.name}

Details

Phone number: ${data.phoneNumber}

Email: ${data.email}

`;
};

start();

function start() {
 const container = document.getElementById('contacts');

```

```

 container.addEventListener('click', onClick);

 onRender();

 function onClick(event) {
 if (event.target.tagName == 'BUTTON') {
 const div = event.target.parentElement.querySelector('.details');
 if (div.style.display == 'block') {
 div.style.display = 'none';
 event.target.textContent = 'Hide Details';
 } else {
 div.style.display = 'block';
 event.target.textContent = 'Show Details';
 }
 }
 }

 function onRender() {
 render(html`${contacts.map(c => contactTemplate(c))}`, container);
 }
}

```

*Option 3 - Подобна задача с декларативен стил, но този път скриваме целия елемент div*

Реално откача div и закача нов див – в жълт/лилав цвет се вижда промяната в elements (inspector) от браузъра. Това според преподавателят е най-добрия вариант, защото ако данните се пазят скрити на страницата, в определени случаи може да ни изиграе лоша шега.

```

import {html, render} from './node_modules/lit-html/lit-html.js';
import {cats as catData} from './catSeeder.js';

//template:
// contains cat info
// has toggle button
const catCard = (cat, showInfo) => html`-
 <div class="info">
 <button @click=${(() => toggleInfo(cat))} class="showBtn" ${cat.info ? 'Hide' : 'Show status code'}></button>
 ${cat.info ? html`<div class="status" id="${cat.id}">
 <h4 class="card-title">Status Code: ${cat.statusCode}</h4>
 <p class="card-text">${cat.statusMessage}</p>
 </div>` : null}
 </div>
`;
}

//start:
//parse imported data
//pass to template

```

```

const root = document.getElementById('allCats');
catData.forEach(c => c.info = false);
update();

function update(params) {
 render(html`${catData.map(catCard)}`, root);
}

function toggleInfo(cat) {
 cat.info = !cat.info;
 update();
}

```

Directives: repeat

- Repeats a **series of values** generated from an iterable, and **updates** those items **efficiently** when the iterable changes:

```

import { repeat } from './node_modules/lit- html/directives/repeat.js';
import { repeat } from 'https://unpkg.com/lit-html/directives/repeat.js?module';

```

```

const myTemplate = () => html`

 ${repeat(items, (i) => i.id, (i, index) => html`- ${index}: ${i.name}`)}

`;

```

Directives: until в комбинация с promise

```

import { until } from './node_modules/lit- html/directives/until.js';
import { until } from 'https://unpkg.com/lit-html/directives/until.js?module';

```

*Вариант 1 - без асинхронна опаковка, директно в темплейта бухаме асинхронната функция delayed.js*

```

import {html, render} from './node_modules/lit-html/lit-html.js';
import {until} from './node_modules/lit-html/directives/until.js';

const asyncTemplate = (dataPromise) => html`

${until(articleTemplate(dataPromise), html`Loading…`)} докато се
 resolve-не dataPromise, показвай Loading ...

`;

const articleTemplate = async (data) => html``

```

```

 <p>${(await data).content}</p>
</article>
`;

export function onUntil() {
 const main = document.querySelector('#content');

 render(asyncTemplate(getData()), main);
}

async function getData() {
 const data = {content: 'Async data'};
 return new Promise(resolve => {
 setTimeout(() => resolve(data), 2000);
 });
}

app.js
import { onUntil } from './delayed.js';
import { html, render } from './node_modules/lit-html/lit-html.js';

import articleTemplate from './templates/articles.js';

start();

async function start() {
 const data = await (await fetch('./data.json')).json(); // това го правихме на 2 реда, сега
 // го правим на един ред
 const main = document.querySelector('main');
 const renderBtn = document.getElementById('renderBtn');
 renderBtn.addEventListener('click', onRender);

 document.getElementById('changeBtn').addEventListener('click', onChange);
 document.getElementById('untilBtn').addEventListener('click', onUntil);
}

```

*Вариант 2 - с асинхронна опаковка, а в темплейта бухаме синхронна/директна функция*

```

delayed.js
import {html, render} from './node_modules/lit-html/lit-html.js';
import {until} from './node_modules/lit-html/directives/until.js';

const asyncTemplate = (dataPromise) => html`

${until(dataPromise, html`Loading...`)}

`;

const articleTemplate = (data) => html`

<p>${(data).content}</p>

`;

```

```

`;

export function onUntil() {
 const main = document.querySelector('#content');

 render(asyncTemplate(resolveTemplate(getData())), main);
}

//wrapper/опаковка
async function resolveTemplate(dataPromise) {
 const data = await dataPromise;

 return articleTemplate(data);
}

async function getData() {
 const data = {content: 'Async data'};
 return new Promise(resolve => {
 setTimeout(() => resolve(data), 2000);
 });
}

```

Directives: cache

```

import {cache} from 'https://unpkg.com/lit-html/directives/cache?module';
cache не работает с промисами!

ctx.renderProp(cache(quizTemplate(ctx.quiz, questions, index)))

```

Directives: if-defined

```

import {ifDefined} from '../../node_modules/lit-html/directives/if-defined.js';
import {ifDefined} from '//unpkg.com/lit-html/directives/if-defined.js?module';

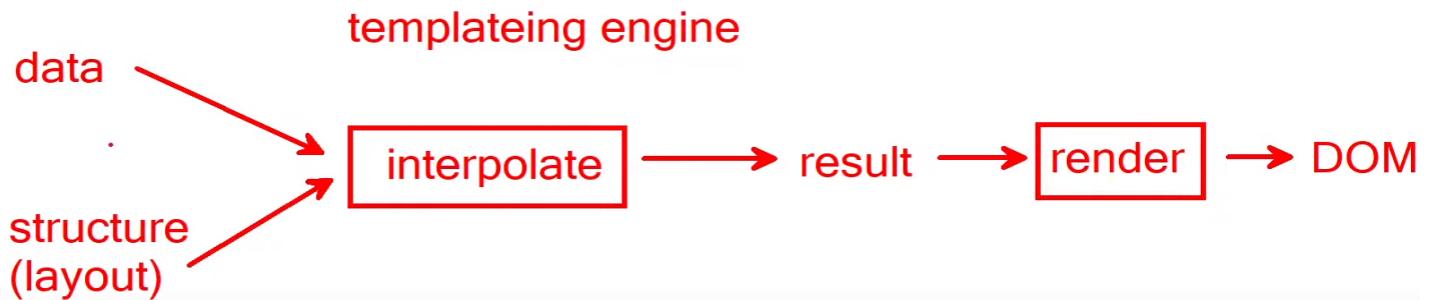
```

```

<div class="mb-3">
 <label for="movie-image-url" class="form-label">Image URL</label>
 <input type="text" class="form-control" name="imageUrl" id="movie-image-url"
.value=${ifDefined(movie.img)}> <!-- with if-defined directive -->
</div>
</pre>

```

## 26.5. Workflow of Client-Side rendering



26.6. Слагаме event listener върху целия form формуляр, не само върху събмит бутона или някой друг бутон

защото може като въведем данни в input и натиснем enter (без да кликнем бутона)

//да генерираме събитие, което няма да може да се хване

```
<form action="#" class="content">
 <label for="towns">Towns</label>
 <input id="towns" name="towns" type="text" />
 <button id="btnLoadTowns">Load</button>
</form>
```

```
import { html, render } from './node_modules/lit-html/lit-html.js';
```

```
const root = document.getElementById('root');
```

// on submit:

// parse input

// render template

// Слагаме event listener върху формулара, защото може като въведем данни и натиснем enter (без да кликнем бутона)

//да генерираме събитие, което няма да може да се хване

```
document.querySelector('form').addEventListener('submit', (event) => {
 event.preventDefault(); //при формулар и anchor слагаме и preventDefault() - да не ходи на нова страница
```

```
 const towns = document.getElementById('towns').value.split(',').map(t => t.trim());
```

```
 const result = listTemplate(towns);
```

```
 render(result, root);
```

```
)
```

```
// template:
```

```
// ul with li for each array item
```

```
const listTemplate = (towns) => html`

 ${towns.map(t => html`${t}`)}

`;
```

## 26.7. Сравнение на onsubmit в HTML и @submit в lit-html

```
<form name="myForm" action="/action_page.php" onsubmit="return validateForm()" method="post">
 Name: <input type="text" name="fname">
 <input type="submit" value="Submit">
</form>

const articleTemplate = (onSubmit, dataEl) =>
 html`

 <article>
 <input type="text" ?disabled=${dataEl.disabled} .value=${dataEl.color}> //добавяме така
 DOM пропърти в lit-html
 <h3>${dataEl.title}</h3>
 <div class="${dataEl.color}">
 <p>${dataEl.content}</p>
 </div>
 <footer>Author: ${dataEl.author}</footer>
 <div class="comments">
 <form @submit=${onSubmit}>
 <textarea name="comment"></textarea>
 <button>Submit comment</button>
 </form>

 ${dataEl.comments.map(commentTemplate)}

 </div>
 </article>`;

```

## 26.8. Сравнение на onclick в HTML и @click в lit-html

## 26.8. Сравнение на onchange в HTML и @change в lit-html

```
<form @change=${onSelect}>
 ${questions.map((q, i) => questionTemplate(q, i, i == currentQuestionIndex))}
</form>
```

## 26.9. Dependency injection с използване на тип callback функция, и с визуализиране на няколко елемента

В scope на текущия файл/модул изнасяме променливата ctx – вече го разгледахме

```
let ctx = null; //за да използваме ctx като през injection-а чрез функцията
showDetailsPage(ctxTarget, id),
```

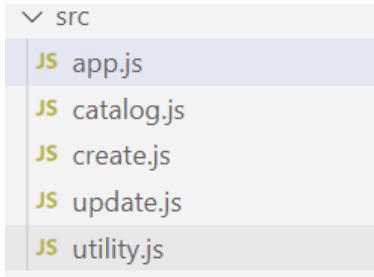
```
export async function showDetailsPage(ctxTarget, id) { //параметрите на goTo функцията
 //директно в scope-а на текущия модул/файл.
```

```

ctx = ctxTarget;
ctx.showSection(section);
loadIdea(id);
}

```

Има и втори вариант за инжектиране – като използваме тип callback функция – при добавяне на книга, тя веднага се визуализира



```

app.js
import { showCatalog } from './catalog.js';
import { showCreate } from './create.js';
import { showUpdate } from './update.js';
import { render } from './utility.js'; //тук сме нахакали заявките и lit-html импортите

```

```

//main module:
// init other modules with dependencies
// - rendering
// - communication between modules
const root = document.body; //тук го използваме без querySelector

```

```

const ctx = {
 updateRender
};

```

```
updateRender();
```

```
//пререднира/изпълнява и трите визуализации на body html структурата
function updateRender() {

```

```

 render([
 showCatalog(ctx),
 showCreate(ctx),
 showUpdate(ctx)],
 root);
}

```

```
create.js
```

```

import {createBook, html} from './utility.js';

//create module:
// control create form

//onSuccess callback
const createTemplate = (onSuccess) => html`
```

```

<form @submit=${ev => onSubmit(ev, onSuccess)} id="add-form">
 <h3>Add book</h3>
 <label>TITLE</label>
 <input type="text" name="title" placeholder="Title...">
 <label>AUTHOR</label>
 <input type="text" name="author" placeholder="Author...">
 <input type="submit" value="Submit">
</form>`;

export function showCreate(ctx){
 //подаваме контекста ctx.updateRender() като callback, и контекста ще стигне до submit-а
 на createTemplate
 if (ctx.book != undefined) {
 return null;
 } else {
 return createTemplate(ctx.updateRender);
 }
}

//onSuccess е callback
async function onSubmit(event, onSuccess) {
 event.preventDefault();
 const formData = new FormData(event.target);
 console.log(event.target);

 const title = formData.get('title').trim();
 const author = formData.get('author').trim();

 await createBook({author, title});

 event.target.reset();
 onSuccess();//изпълняване на callback функцията
}

```

```

catalog.js
import {deleteBook, getBooks, html, until} from './utility.js';

//list module:
// display list of books
// control books(edit, delete)

const catalogTemplate = (booksPromise) => html`


```

```

<tbody>
 ${until(booksPromise, html`<tr><td colSpan = "3">Loading…</td></tr>`}
</tbody>
</table>`;

const bookRow = (book, onEdit, onDelete) => html`
<tr>
 <td>${book.title}</td>
 <td>${book.author}</td>
 <td>
 <button @click=${onEdit}>Edit</button>
 <button @click=${onDelete}>Delete</button>
 </td>
</tr>`;

export function showCatalog(ctx){
 return catalogTemplate(loadBooks(ctx));
}

async function loadBooks(ctx) {
 const data = await getBooks();

 const books = Object.entries(data).map(([k, v]) => Object.assign(v, {_id: k}));
 console.log(books);

 return books.map(bk => bookRow(bk, toggleEditor.bind(null, bk, ctx), onDelete.bind(null,
bk._id, ctx)));
}

function toggleEditor(book, ctx) {
 ctx.book = book; //задаваме ново текущо свойство на ctx - да има book
 ctx.updateRender();
}

async function onDelete(id, ctx) {
 await deleteBook(id);
 ctx.updateRender();
}

update.js
import {html, updateBook} from './utility.js'; // update module:// control edit form

B lit-html name
const updateTemplate = (book, ctx) => html`
<form @submit=${ev => onSubmit(ev, ctx)} id="edit-form">
 <input type="hidden" name="id" .value=${book._id}>
 <h3>Edit book</h3>
 <label>TITLE</label>
 <input type="text" name="title" placeholder="Title..." .value=${book.title}>
 <label>AUTHOR</label>
 <input type="text" name="author" placeholder="Author..." .value=${book.author}>

```

```

<input type="submit" value="Save">
</form>
`;

//подаваме 2 callback параметъра - един за данните за книгата, и втори контекста
ctx.updateRender()и -двета ще стигнат при изпълнение submit-а на updateTemplate
export function showUpdate(ctx){
 //как да разберем дали ще показваме create (Book) или update (Book)
 if (ctx.book == undefined) {return null;} else { return updateTemplate(ctx.book, ctx);
 }
}

async function onSubmit(event, ctx) {
 debugger;
 event.preventDefault();
 const formData = new FormData(event.target);

 const id = formData.get('id'); //все едно викаме от формуляра name, което в
случая е с име name id
 const title = formData.get('title').trim(); //все едно викаме от формуляра name, което в
случая е title
 const author = formData.get('author').trim(); //все едно викаме от формуляра name, което
в случая е author

 await updateBook(id, {author, title});

 event.target.reset();
 delete ctx.book; //трием пропъртито да го няма
 ctx.updateRender(); //изпълняване на callback функцията
}

```

## 27. Browser Routing

Рутиране и маршрутизация в браузъра

Има и рутиране в сървъра

### 27.1. Routing Concepts - Types of Navigation in Web Applications

Multi Page Applications

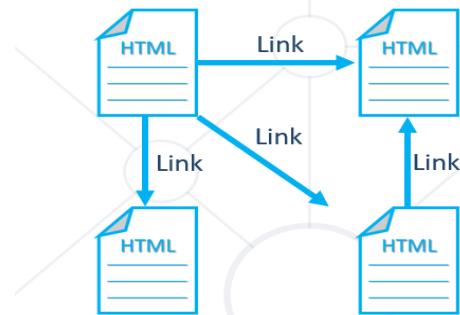
- **Reloads** the entire page
- **Displays** the **new page** when a user interacts with the web app
- When a data is exchanged, a **new page** is **requested** from the server to display in the web browser

Single Page Applications

- Web apps that load a **single HTML file**
- SPAs use **AJAX** and **HTML5** to create fluid and responsive Web apps
- **No constant page reloads**

## Navigation Types

### ▪ Standard Navigation



- Navigation using **Routing** - allows navigation, **without reloading** the page



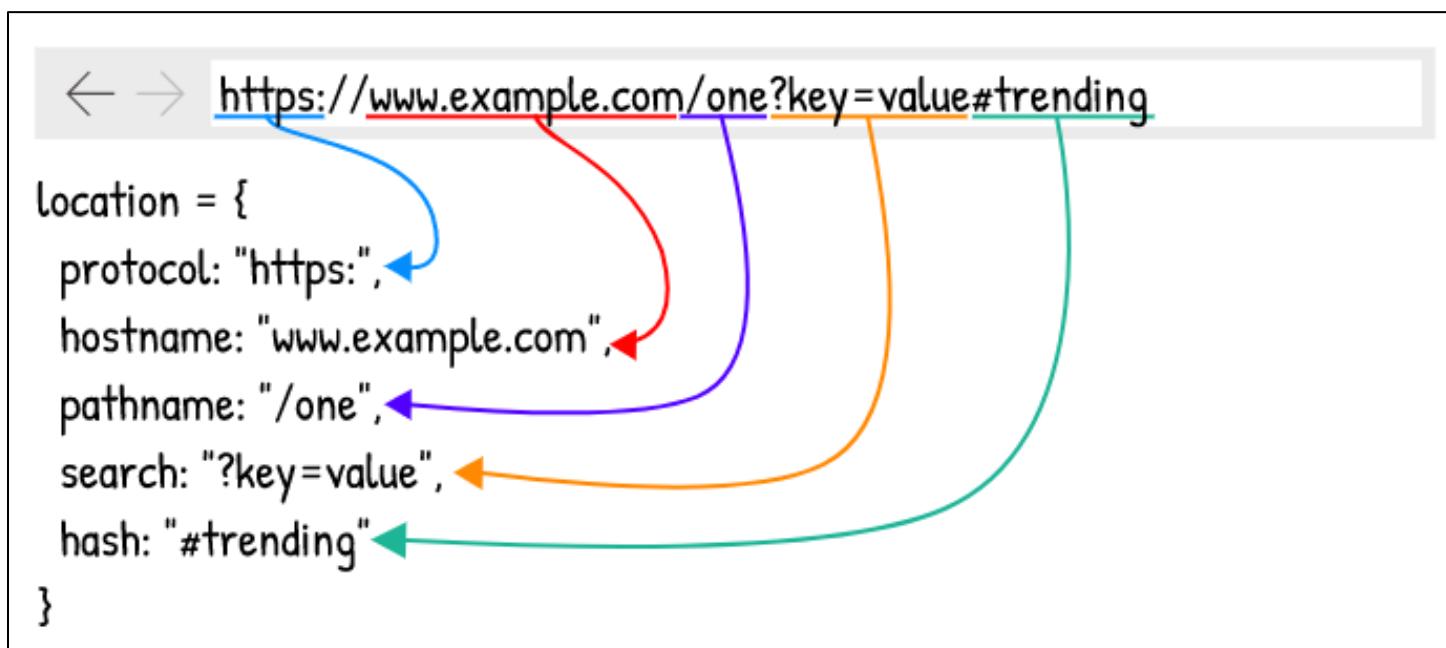
## 27.2. Client-Side Routing - Navigation for Single Page Apps

### How Routers Work

- A **Router** tells the Application **when** the location **has changed**
  - E.g. when the user manually **enters an URL address in the browser**
- Conversely(противоположното), a **change in content** reflects the router and he changes the **URL address bar**
  - E.g. when the user **clicks on a link**



### Location



## Protocol

http или https от secured

Host name – името на сайта

Pathname/ресурс – все едно са директории с файлове

**/catalog/kitchen/1234** - 1234 е id на конкретен продукт

Query params/Querystring/ Query/ Search params - след питанката (?), частта се казва:

**Search**

**Search params**

**Querystring**

**Query**

**Query params**

Hash / Fragment/ Id - след диеса (#), частта се казва:

**Hash**

**Fragment**

**Id**

Всеки фрагмент има id, и ние пишем id-то на фрагмента, към който искаме да отидем. Т.нар. Deep Link

**Тази част след # не отива като информация към сървъра, а служи за информация на Browser-а какво да визуализира – това е много важно! Всичко преди # отива към сървъра като заявка.**

## Query Parameters

- Allow for application state to be serialized into the URL
- Represented by a series of key-value pairs, separated by & амперсант

▪ Example: **search=js+advanced&opCourses=true**

**First parameter**

**Second parameter**

Като обектите, само че разделени с равно

Ключ search = value js+advanced

Ключ opCourses = value true

```

catalog.js:3
Context {page: Page, canonicalPath: '/?page=3', path: '/page=3', title: 'Furniture', state: {...}, ...}
 ▾ page: Page {callbacks: Array(8), exits: Array(0), curr
 ▾ params:
 ▶ [[Prototype]]: Object
 path: "/?page=3"
 pathname: "/"
 queryString: "page=3" highlighted
 ▶ renderProp: (content) => render(content, root)
 routePath: "/"
 ▶ state: {path: '/?page=3'}
 title: "Furniture"
 ▶ updateUserNav: f updateUserNav()

```

#### Кодиране и декодиране на URL елементи

```

> encodeURIComponent('JS Applications')
< 'JS%20Applications'
> decodeURIComponent('JS%20Applications');
< 'JS Applications'

>> encodeURIComponent('js applications')
< "js%20applications"
>> decodeURIComponent("js%20applications")
< "js applications"
>> encodeURIComponent('{"userId": "q9345n93847j9ce9rt"}')
< "%7B%22userId%22%3A%20%22q9345n93847j9ce9rt%22%7D"

```

**В URL-а не може да имаме space или други контролни символи. И затова контролни символи правим url encoding:**

Пример:

+ или %20 – space интервал  
 %3D – равно  
 %22 – кавичка

#### Common use cases of query

- **Serializing additional application state**
- Representing the current page number **in a paginated collection**
- Filter criteria
- Sorting criteria
- Search criteria

- Select properties of an item

Pagination със сървър <http://localhost:3030/data/records>

Инструкции за ползване на самия сървър - <https://github.com/softuni-practice-server/softuni-practice-server/blob/master/COLLECTIONS.md#pagination>

`offset={skip}&pageSize={take}`

GET /data/recipes?offset=10&pageSize=5 - пропусни първите 10 записи, и дай 5те следващи тъй като на една страница са 5 записи.

Формулата за offset е: `offset = (страница - 1) * pageSize;`

**offset 1 – 1 = 0 \* 5 = 0**

**offset 2 – 1 = 1 \* 5 = 5**

**offset 3 – 1 = 2 \* 5 = 10**

<http://localhost:3030/data/records> - връща 10 записи по подразбиране

<http://localhost:3030/data/records?pageSize=3> - връща само 3 записи

<http://localhost:3030/data/records?offset=3&pageSize=3> - това на практика е втора страница

Sorting със сървър <http://localhost:3030/data>

<https://github.com/softuni-practice-server/softuni-practice-server/blob/master/COLLECTIONS.md#sorting>

To sort by creation time, newest first (descending):

(unencoded) /data/recipes?sortBy=\_createdOn desc

GET /data/recipes?sortBy=\_createdOn%20desc

(encoded) [http://localhost:3030/data/recipes?sortBy=\\_createdOn%20desc](http://localhost:3030/data/recipes?sortBy=_createdOn%20desc)

Searching със сървър <http://localhost:3030/data>

<https://github.com/softuni-practice-server/softuni-practice-server/blob/master/COLLECTIONS.md#search>

6 вида търсене предлага сървъра

- Чрез директно съвпадение

(unencoded) /data/comments?where=recipeId="8f414b4f-ab39-4d36-bedb-2ad69da9c830"

- Compare numbers с използването на val пропърти на обектите в базата данни на сървъра:

`where=val >= 5`

(unencoded) <http://localhost:3030/data/records?where=val=3>

<http://localhost:3030/data/records?where=val%3d3>

The screenshot shows a JSON viewer interface with the URL `localhost:3030/data/records?where=val%3d3`. The data is displayed in a tree structure:

```

0:
 name: "John6"
 val: 3
 _createdOn: 1613551388753
 _id: "i06"

1:
 name: "John7"
 val: 3
 _createdOn: 1613551388763
 _id: "i07"

```

The `val` property for both records is highlighted with a red underline, indicating it's the search criteria used in the query.

- Чрез String contents (case-insensitive):  
 where=title LIKE "lasagna"  
 (unencoded) <http://localhost:3030/data/records?where=name%20LIKE%20%22n1%22>

Selecting със сървър <http://localhost:3030/data>  
**topicById: '/data/topics?select=\_id,title',**

Комбинация от Selecting и команда load(relations) със сървър <http://localhost:3030/data>

```
const endpoints = {
 topics:
 `/data/topics?select=_id,title,_ownerId&load=${encodeURIComponent('author=_ownerId:users')}`,
 //query - идва от функционалността на сървъра
 topicCount: `/data/topics?count`, //query - идва от функционалността на сървъра
 topicById: (id) =>
 `/data/topics/${id}?load=${encodeURIComponent('author=_ownerId:users')}`, //query - идва от
 //функционалността на сървъра
 createTopic: `/data/topics`
};
```

Hash-based Routing / хеш

- Using the **#hash** part of the URL to simulate different content
- The routing is possible because changes in the hash **don't trigger page reload**

### Example

```
» window.location
← ▶ Location https://en.wikipedia.org/wiki/JavaScript
X JavaScript - Wikipedia × +
← → ⌂ ⚡ https://en.wikipedia.org/wiki/JavaScript
```

Това зарежда направо нова страница:

```
window.location = 'https://en.wikipedia.org/wiki/ECMAScript' ;
```

Изпълнявайки това действие, страницата не се презарежда

```
» window.location.hash = '#Trademark'
← "#Trademark"
D 🔒 https://en.wikipedia.org/wiki/JavaScript#Trademark
```

Така **НЕ** се прави

Няма причина да хващаме href при положение че имаме .hash пропърти на window.location

Extracting the hash from the entire URL

```
let hash = window.location.href.split('#')[1] || '';
```

Changing the path

```
//задай нов hash
let changePath = function (path) {
 let currentPath = window.location.href;
 window.location.href = currentPath.replace(/#(.*)$/, '') + '#' + path;
}
```

Processing hashchange events

- Using an event handler:

```
window.onhashchange = funcRef;
```

- Using an HTML event handler:

```
<body onhashchange="funcRef()">
```

- Using an event listener:

```
window.addEventListener("hashchange", funcRef, false);
```

Страницата не се презарежда

Хванахме смяната на #hasha – при ръчно писане в URL-а, както и при въвеждане на команда  
changeHash('about Lets Say')

- ⓘ 127.0.0.1:5500/index.html#home
- ⓘ 127.0.0.1:5500/index.html#catalog

```
<body>
 <main>
 <h1>Hash Routing Demo</h1>
 Home <!--Когато щракнем на линка, диес home става новия hash-->
 Catalog <!--Хешовете винаги се добавят в href атрибуута-->
 About
 </main>

</body>

<script>
 window.addEventListener('hashchange', (event) => {
 console.log('Hash has changed');
 console.log(event);
 })

 function changeHash(hash) {
 window.location.hash = hash; //добавя нов
 }
</script>
```

```

<body>
 <h1>Hash Routing Demo</h1>

 <nav>
 Home
 <!-->
 Catalog
 About
 </nav>

 <main>
 </main>

</body>

```

*Hash-demo - Така се прави*

```

<script>
 const views = {
 '#home': () => '<h2>Home page </h2><p>Welcome to our site</p>',
 '#catalog': () => '<h2>Catalog</h2><p>List of items</p>',
 '#about': () => '<h2>About us</h2><p>Contact: +15184388 215</p>'
 };

 const main = document.querySelector('main');

 window.addEventListener('hashchange', onHashChange);

 //we load the initial info/home info
 onHashChange();

 function onHashChange() {
 const hash = window.location.hash;

 const view = views[hash];

 if (typeof view == 'function') {
 main.innerHTML = view(); //тук използваме е() функцията за построяване на DOM
 структура или client-side rendering (например с lit-html)
 } else {
 main.innerHTML = '<h2>404</h2><p>Page Not Found</p>';
 }
 }
</script>

```

## Push-Based Routing

- You can actually surface real **server-side data** to support things like SEO and Facebook Open Graph
- It helps with **analytics**
- It helps fix **hash tag issues**
- You can actually use hash tag for what it was meant for, **deep linking** to sections of long pages

## History API

- Provides access to the browser's history through the **history** object
- **HTML5** introduced the **history.pushState()** and **history.replaceState()**
  - They allow you to add and modify **history entries**
  - These methods work in conjunction with the **popstate** event

```
> window.history
< ▼History {length: 4, scrollRestoration: 'auto', state: null} ⓘ
 length: 4
 ...
 - - - - -
```

Отиваме на предишно състояние – едно назад

```
> history.back()
```

Едно напред с метода `history.forward()`

Едно напред с метода `go()`

```
> history.go(1)
< undefined
```

Две назад с метода `go()`

```
> history.go(-2)
< undefined
```

### *The PushState() Method*

- Adds new object to the history of the browser
- Takes three parameters:
  - **State**
    - Object which is associated with the new history entry
  - **Title**
    - Browsers currently ignore this parameter
  - **URL**
    - The new history entry's URL is given by this parameter
    - It must be of the **same origin** as the current URL

```
> history.pushState({}, '', '#catalog')
```

Текущата страница става последната заредена страница става с `href/hash #catalog` в списъка История на заредените страници/инфота. Нямаме опция да дадем на `redo/forward` от `#catalog`

**The PushState() Method** не тригерва event-а ‘hashchange’ – т.е. в URL се променя, но не и съдържанието на страницата.

#### *The ReplaceState() Method*

- **Modifies the current history entry (which is the topmost)** instead of creating a new one
- It is particularly useful when you want to update the **state object** or **URL** of the current history entry

```
let stateObj = { facNum: "56789123" };
history.pushState(stateObj, "", "student.html");
history.replaceState(stateObj, "", "newStudent.html");
```

› `history.replaceState({}, '', '#about')`

Текущата страница от списъка История на заредените страници/инфота става с хеш #about.

**The ReaplceState() Method** не тригерва event-а ‘hashchange’ – т.е. в URL се променя, но не и съдържанието на страницата

#### *The Popstate Event*

- Dispatched to the window every time the active history entry changes
- If the history entry being activated was created by a call to **pushState** or affected by a call to **replaceState**,
- The **popstate** event's **state property** contains a copy of the history entry's state object
- You can read the state of the current history entry without waiting for a **popstate** event using the **history.state property**
- Или с други думи – това което е активно в URL-а като hash или като pathname, то да се визуализира и на страницата

Реално, за history не трябва да ползваме hash, а само pathname. Така е правилно. И адресите изглеждат добре. И информацията ще се предава на сървъра. А при използване на hash, сървъра няма да получи инфо какво да изпрати като инфо за зареждане при клиента. Също така ако искаме да share-нем страница, и страницата няма pathname, то получателят на share-ването като го пусне на своя браузър, ще получи целия сайт първоначален, а не конкретната визуализация, която сме искали да share-нем.

#### *History-demo*

При стартиране ще видим какво е URL-то и хеша, и оттам нататък при навигация, ще го направим да не се презарежда страницата. Сега, ако някой от навигираните линкове ако се копира на друг бразуър прозорец на същия или на друг компютър – това е друга тема.

**a -> addEventLister -> preventDefault()**



**new content**

```
<body>
 <h1>History Demo</h1>
```

```

<nav>
 Home
 Catalog
 About
</nav>

<main>
</main>

</body>

<script>
 //Като се връщаме напред/назад в history-то, визуализирай съответната информация на страницата
 window.addEventListener('popstate', showContent); //trigger the show content on the page after we have pushState (or replaceState) in history

 const views = {
 '/home': () => '<h2>Home page </h2><p>Welcome to our site</p>',
 '/catalog': () => '<h2>Catalog</h2><p>List of items</p>',
 '/about': () => '<h2>About us</h2><p>Contact: +15184388 215</p>'
 };

 const main = document.querySelector('main');
 document.body.addEventListener('click', ev => {
 //Когато потребителят цъка на линкове
 if (ev.target.tagName == 'A') {
 ev.preventDefault(); //недей презареждай страницата
 history.pushState({}, '', ev.target.href); //вземи href-а(който в случая е pathname) от html от дадения линк и го сложи в URL-а, както и в history-то
 showContent(); //покажи текущото инфо от страницата
 }
 });
}

//When we start the page
showContent();

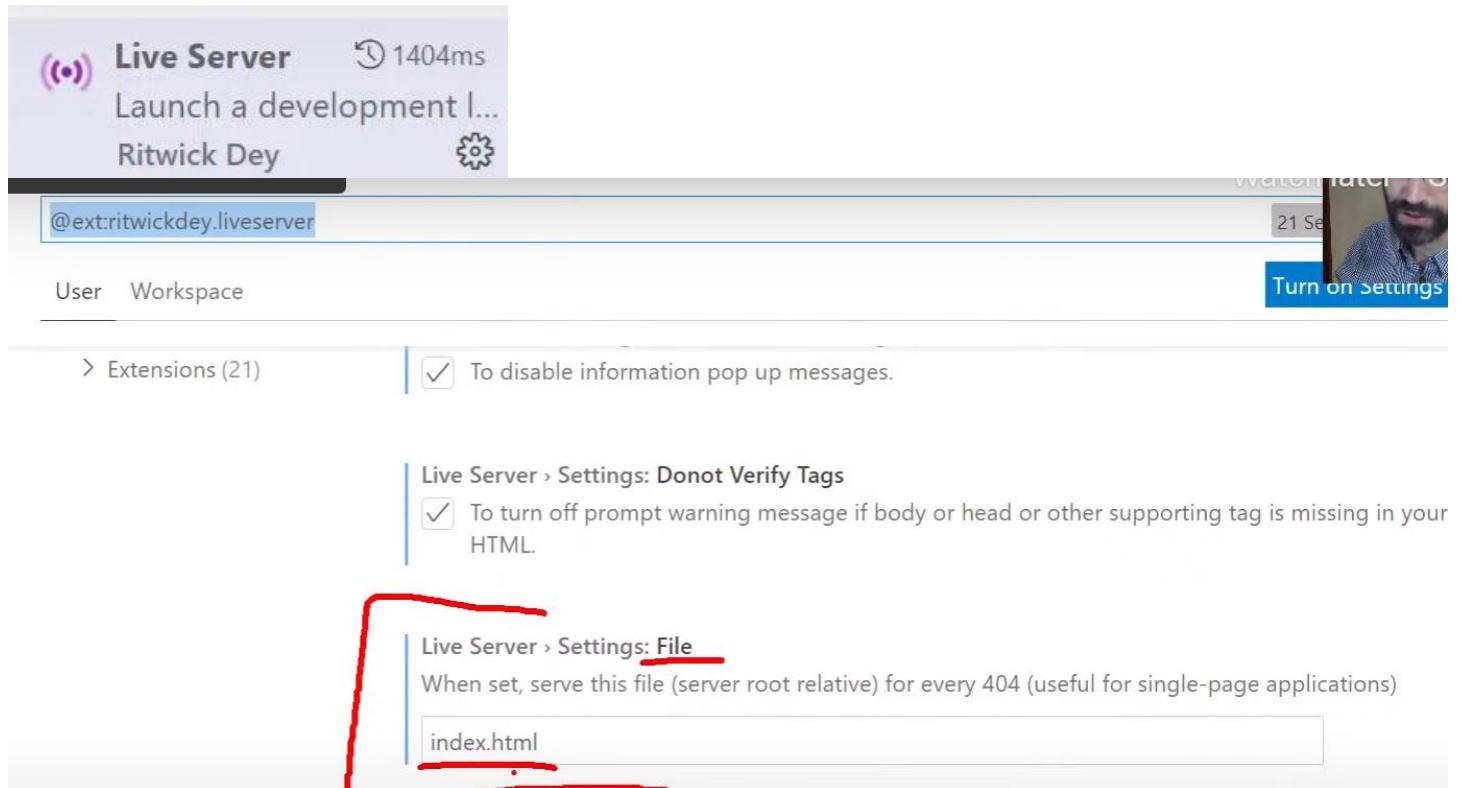
function showContent() {
 const path = window.location.pathname;

 const view = views[path];

 if (typeof view == 'function') {
 main.innerHTML = view();
 } else {
 main.innerHTML = '<h2>404</h2><p>Page Not Found</p>';
 }
}
</script>

```

### 27.3. Настройка на Live Server да ни зарежда SPA - index.html като старт на сайта



Help settings.json - history-demo - Visual Studio Code

JS app.js index.html Extension: Live Server Settings settings.json 1 X

```
C: > Users > svilk > AppData > Roaming > Code > User > {} settings.json > liveServer.settings.json
```

```
19 "workbench.iconTheme": "material-icon-theme",
20 "[javascript)": {
21 "editor.defaultFormatter": "vscode.typescript-language-features"
22 },
23 "prettier.singleQuote": true,
24 "[html)": {
25 "editor.defaultFormatter": "vscode.html-language-features"
26 },
27 "prettier.tabWidth": 4,
28 "editor.tabCompletion": "on",
29 "workbench.colorTheme": "Default Light+",
30 "workbench.editorAssociations": {
31 "*.js": "default"
32 },
33 "workbench.startupEditor": "none",
34 "liveServer.settings.donotShowInfoMsg": true,
35 "javascript.updateImportsOnFileMove.enabled": "always",
36 "liveServer.settings.file": "index.html"
37 }
```

./ - текуща папка

../ - една папка по-горе

/ - абсолютна директория/папка – спрямо старта на нашето приложение

## 27.4. External Routing Library - using page.js for Single-Page Routing

### Using **page.js** for Single-Page Routing

What is page.js?

- Compact **client-side router**
  - **Small size** – 1.2KB
- **Syntax** inspired by Express (back-end framework)
- Supports:
  - Automatic **link binding**
  - URL glob **matching**
  - **Parameters**
  - Plugins

### Getting Started

- Installation via **npm** package: На изпита се инсталира локално

**npm init -y**

**npm install page**

```
import page from '../node_modules/page/page.mjs'; //Ecma script 6 стандартът работи
```

- Direct **import** from online **CDN** (no installation):

```
import page from "//unpkg.com/page/page.mjs";
```

- Basic Usage:

Първият параметър е адреса на нещото, а втория параметър е нещото което се случва

```
page('/', index); // Register home route
page('*', notfound); // Register catch-all (404)
page.start(); // Activate router
```

- Documentation: <https://github.com/visionmedia/page.js>

### Basic Routing

- Routes are registered via **match pattern** and **callback** - първият параметър е адреса на нещото, а втория параметър е нещото което се случва

```
page('/catalog', catalogView);
```

- Match pattern can be string, URL glob or RegExp
- The **route handler** (callback) will receive two parameters
  - **context** object with information about **parameters** and **state**
  - **next** callback, used when **chaining** route handlers

```
function catalogPage(ctx, next) {
// fetch data, render template, handle form, etc.
 console.log(ctx);
 console.log(next);
 main.innerHTML = '<h2>Home page </h2><p>Welcome to our site</p>';
}
```

## URL Parameters

- URL **glob patterns** can match **dynamic** parts of the URL
  - E.g., category name, product ID, user page, etc.

```
page('/catalog/:id', detailsView);
// match any route, following /catalog
```

- The URL **parameter** can be accessed from the **context**

```
Function detailsView(ctx, next) {
 console.log(ctx.params.id);
}
```

→ C ⓘ 127.0.0.1:5500/details/34a1cab1-81f1-47e5-aec3-ab6c9810efe1

```
Context {page: Page, canonicalPath: '/details/34a1cab1-81f1-47e5-aec3-ab6c9810efe1',
 h: '/details/34a1cab1-81f1-47e5-aec3-ab6c9810efe1', title: 'Team Manager', state:
 {...}, ...} ⓘ
 canonicalPath: "/details/34a1cab1-81f1-47e5-aec3-ab6c9810efe1"
 hash: ""
 init: true
 ▶ page: Page {callbacks: Array(14), exits: Array(0), current: '/details/34a1cab1-81f1'
 ▶ params:
 id: "34a1cab1-81f1-47e5-aec3-ab6c9810efe1"
 ► [[Prototype]]: Object
 path: "/details/34a1cab1-81f1-47e5-aec3-ab6c9810efe1"
 pathname: "/details/34a1cab1-81f1-47e5-aec3-ab6c9810efe1"
 querystring: ""
 ► renderProp: (content) => render(content, main)
 routePath: "/details/:id"
 ► setUserNavProp: f setUserNav()
 ► state: {path: '/details/34a1cab1-81f1-47e5-aec3-ab6c9810efe1'}
 title: "Team Manager"
```

- **Multiple parameters** can be captured

```
page('/:category/:id', detailsView); //ако го кръстим тук с различно име от id ще гърми -
 защото ctx.params.id търси
```

## Programmatic Redirect

- Setup automatic redirect upon visit

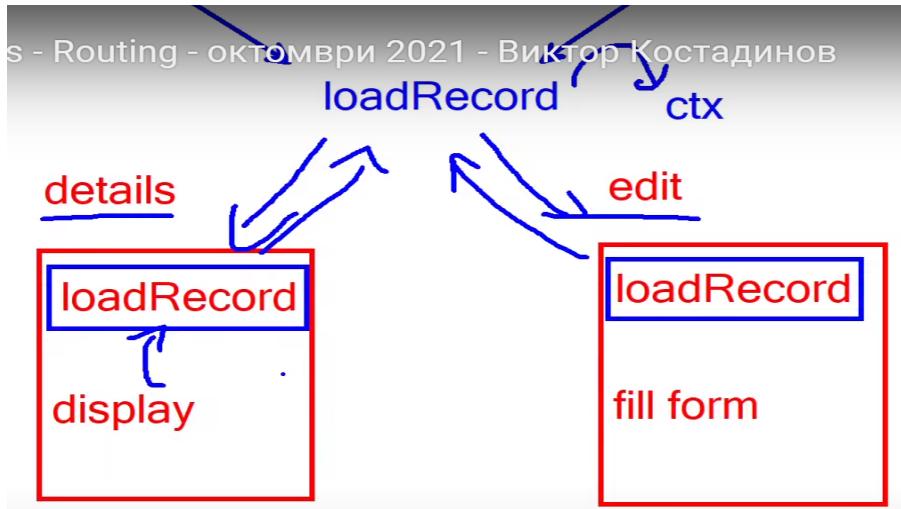
Програмистична навигация с редирект/ротиране

```
page.redirect('/home', '/catalog');
// navigating to /home will be redirected to /catalog
```

- Navigate to a page programatically

```
page.redirect('/login');
```

## Chaining Route Handlers



Дисплея или fill form може да види кой е предишния eventHandler, който се е извикал.

- Route handlers can be **chained**

```
page('/catalog/:id', loadData, detailsView);
```

- Practical when **separating concerns**

- E.g., **fetch** remote data in one handler and **render** in another

- Add values to the **context**, to share them **across handlers**

```
async function loadData(ctx, next) {
 const data = await fetchProduct(ctx.params.id);
 ctx.product = data;
 next();
}
```

## Page-demo

Грубата версия

```
import page from '//unpkg.com/page/page.mjs';
```

```
const views = {
 '/home': () => '<h2>Home page </h2><p>Welcome to our site</p>',
 '/catalog': () => '<h2>Catalog</h2><p>List of items</p>',
 '/catalog/kitchens': () => '<h2>Kitchen Equipments</h2><p>List of items</p>',
 '/about': () => '<h2>About us</h2><p>Contact: +15184388 215</p>'
};
```

```
const main = document.querySelector('main');
```

//тук извикваме с lambda функция, а production версията извикваме по референция(което е почти същото, но с референция си вземаме и контекста)

```
page('/home', () => showContent('/home'));
page('/catalog', () => showContent('/catalog'));
page('/catalog/kitchens', () => showContent('/catalog/kitchens'));
page('/about', () => showContent('/about'));
```

```

page.start(); //стартираме го

function showContent(name) {
 const view = views[name];

 if (typeof view == 'function') {
 main.innerHTML = view();
 } else {
 main.innerHTML = '<h2>404</h2><p>Page Not Found</p>';
 }
}

```

*Production версията*

```

import page from '//unpkg.com/page/page.mjs';

const main = document.querySelector('main');

function HomePage(ctx, next) {
 console.log(ctx);
 console.log(next);
 main.innerHTML = '<h2>Home page </h2><p>Welcome to our site</p>';
}

function catalogPage() {
 main.innerHTML = '<h2>Catalog</h2><p>List of items</p> Product';
 ';
}

function detailsPage(ctx) {
 console.log(ctx);
 Context {page: Page, canonicalPath
 o', state: {...}, ...} ⓘ
 canonicalPath: "/catalog/1258"
 hash: ""
 init: true
 ▶ page: Page {callbacks: Array(4),
 ▶ params:
 id: "1258"
 ▶ [[Prototype]]: Object
 path: "/catalog/1258"
 pathname: "/catalog/1258"
 querystring: ""
 routePath: "/catalog/:id"
 ▶ state: {path: '/catalog/1258'}
 title: "Page Demo"
 ▶ [[Prototype]]: Object
}

```

127.0.0.1:5500/catalog/1258

Bookmarks Dict Online platforms SoftUni LCW other SOFT Acade...

Elements Console Sources Network Performance Memory >

top Filter Default levels

```

Context {page: Page, canonicalPath: '/catalog/1258', path: '/catalog/1258',
o', state: {...}, ...} ⓘ
 canonicalPath: "/catalog/1258"
 hash: ""
 init: true
▶ page: Page {callbacks: Array(4), exits: Array(0), current: '/catalog/1258'}
 ▶ params:
 productNumber: "1258"

```

127.0.0.1:5500/catalog/kitchens

```

Context {page: Page, canonicalPath: '/catalog/kitchens', path: '/catalog/kitchens', title:
age Demo', state: {...}, ...} ⓘ
 canonicalPath: "/catalog/kitchens"
 hash: ""
▶ page: Page {callbacks: Array(4), exits: Array(0), current: '/catalog/kitchens', len: 1,
 ▶ params:
 productNumber: "kitchens"
 main.innerHTML = '<h2>Product</h2><p>Product details</p><button>Buy now</button>';

document.querySelector('button').addEventListener('click', () => {
 page.redirect('/checkout');
});
}

function checkoutPage() {
 main.innerHTML = '<h2>Cart details</h2><p>Products in cart</p>';
}

function aboutPage() {
 main.innerHTML = '<h2>About us</h2><p>Contact: +15184388 215</p>';
}

// С референция си вземаме и контекста
page('/home', HomePage);
page('/catalog', catalogPage);
page('/catalog/:productNumber', detailsPage); // хвани адрес /catalog и след него има нещо
еще. Това означава :id (:id е placeholder като може да го кръстим и по друг начин, например
productNumber и т.н.)

page('/catalog/:category/:productNumber', detailsPage); // вложени множество параметри – за
да го хванем в параметри, то трябва да използваме : две точки. Ако използваме * звезда, ще
хване всичко, но няма да влезе в параметри

```

```

Context {page: Page, canonicalPath: '/catalog/kitchens/kitc123', path:
 'kitc123', title: 'Page Demo', state: {...}, ...} ⓘ
 canonicalPath: "/catalog/kitchens/kitc123"
 hash: ""
 init: true
 ▶ page: Page {callbacks: Array(4), exits: Array(0), current: '/catalog'}
 ▶ params:
 category: "kitchens"
 productNumber: "kitc123"

page('/checkout', checkoutPage);
page('/about', aboutPage);

page.redirect('/', '/home'); //ако въведем главния адрес без / или главния адрес с / само, то
отиди на homepage

```

page.start(); //стартираме го, то само ще си потърси един вид функцията showContent активирана от eventListener-а или при стартиране на страницата

## 27.5. Симулация на page.js

```

const registry = {};

function registerHandler(url, ...handlers) {
 registry[url] = handlers;
}

function handleRequest(handlers) {
 const ctx = {};

 callNext();
}

function callNext() {
 const handler = handlers.shift();

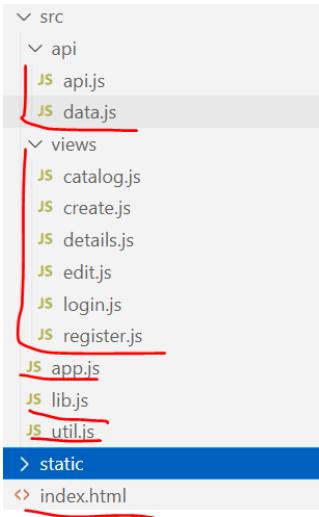
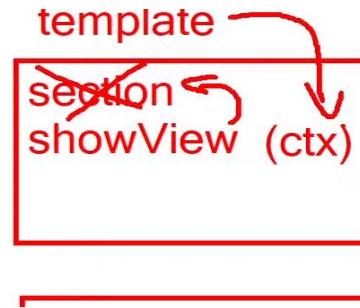
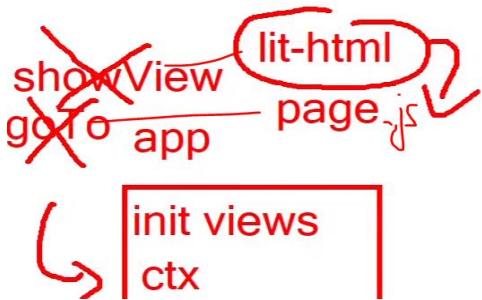
 if (typeof handler == 'function') {
 handler(ctx, callNext);
 }
}

```

!!!!This one – with async/await - 27.6. Demo Movies с използване на page.js и lit-html – dependency injection

Реално, темплейтите скрити накрая на html-а, сега ги слагаме/бухаме в lit-html-а.

Контекста, който идва е от page.js



index.html

```
<head>
<script src="/src/app.js" type="module"></script>
</head>

<body>
 <div class="container" id="container">
 <nav class="navbar navbar-expand-lg navbar-dark bg-dark ">
 Movies
 <ul class="navbar-nav ml-auto ">
 <li class="nav-item user">
 Welcome, email

 <li class="nav-item user">
 Logout

 <li class="nav-item guest">
 Login

 <li class="nav-item guest">
 Register

 </nav>
 </div>
```

```

<main></main>

</div>
</body>

utils.js
import { getMovieById } from "./api/data.js";

//вдигаме нивото на абстракция, за да имаме по-лесен достъп до някои от функциите свързани
//със заявки
export function getUserData() {
 return JSON.parse(sessionStorage.getItem('userData'));
}

//вдигаме нивото на абстракция, за да имаме по-лесен достъп до някои от функциите свързани
//със заявки
export function setUserData(data) {
 sessionStorage.setItem('userData', JSON.stringify(data));
}

//вдигаме нивото на абстракция, за да имаме по-лесен достъп до някои от функциите свързани
//със заявки
export function clearUserData() {
 sessionStorage.removeItem('userData');
}

//middle-ware = посредник - контекста от едно събитие с next се предава същия контекст и на
//следващото събитие - идва от page.js
//като модифицираме ctx тук, то наследника по веригата ще го види
export function loadMovie(ctx, next) {
 const moviePromise = getMovieById(ctx.params.id); //не сме го await-нали нарочно
 ctx.moviePromise = moviePromise; //добавяме свойство филм
 next();
}

api.js
import { clearUserData, getUserData, setUserData } from "../util.js";

const host = 'http://localhost:3030';

async function request(url, options) {
 try {
 const response = await fetch(host + url, options);

 if (response.ok != true) {
 if (response.status == 403) {
 clearUserData(); //from utils.js
 }
 }
 }
}

```

```
 const error = await response.json();
 throw new Error(error.message);
 }

 /* try {
 return await response.json();
 } catch (err) {
 return response;
 } */

 // if (response.status == 204) {
 // return response;
 // } else {
 // return response.json();
 // }

 if (response.status == 204) {
 return response;
 } else {
 return response.json();
 }

} catch (err) {
 alert(err.message);
 throw err;
}
}

function createOptions(method = 'get', data) {
 const options = {
 method,
 headers: {}
 };

 if (data != undefined) { //имаме данни по заявката
 options.headers['Content-Type'] = 'application/json';
 options.body = JSON.stringify(data);
 }

 const userData = getUserData(); //from utils.js
 if (userData != null) {
 options.headers['X-Authorization'] = userData.token;
 }

 return options;
}

export async function get(url) {
```

```
 return request(url, createOptions());
}

export async function post(url, data) {
 return request(url, createOptions('post', data));
}

export async function put(url, data) {
 return request(url, createOptions('put', data));
}

export async function del(url) {
 return request(url, createOptions('delete'));
}

export async function login(email, password) {
 const result = await post('/users/login', {email, password});
 //Оттук надолу заявката ще се изпълни успешно само ако сме минали през горния ред успешно

 const userData = {
 email: result.email,
 id: result._id,
 token: result.accessToken
 };

 setUserData(userData); //from utils.js
}

export async function register(email, password) {
 const result = await post('/users/register', {email, password});
 //Оттук надолу заявката ще се изпълни успешно само ако сме минали през горния ред успешно

 const userData = {
 email: result.email,
 id: result._id,
 token: result.accessToken
 };

 setUserData(userData); //from utils.js
}

export async function logout(){
 await get('/users/logout');
 clearUserData(); //from utils.js

 //Или асинхронно, поред едно след друго се изпълнява - виж app.js
 export async function logout(){
 get('/users/logout');
 clearUserData(); //from utils.js
 }
}
```

```
data.js
import * as api from './api.js';

export const login = api.login;
export const register = api.register;
export const logout = api.logout;

//urls
const endpoints = {
 allMovies: '/data/movies',
 movieById: '/data/movies/'
};

export async function getAllMovies() {
 return api.get(endpoints.allMovies);
}

export async function getMovieById(id) {
 return api.get(endpoints.movieById + id);
}
```

```
lib.js
import {html, render} from '../node_modules/lit-html/lit-html.js';
import page from '../node_modules/page/page.mjs'; //Ecma script 6 стандартът работи
import {until} from '../node_modules/lit-html/directives/until.js';

export {
 html,
 render,
 page,
 until
}
```

```
app.js
import { logout } from "./api/data.js";
import { page, render } from './lib.js';
import { getUserData, loadMovie } from './util.js';
import { catalogPage } from './views/catalog.js';
import { createPage } from './views/create.js';
import { detailsPage } from './views/details.js';
import { editPage } from './views/edit.js';
import { loginPage } from './views/login.js';
import { registerPage } from './views/register.js';
```

```
const root = document.querySelector('main');
document.getElementById('logoutBtn').addEventListener('click', onLogout);
```

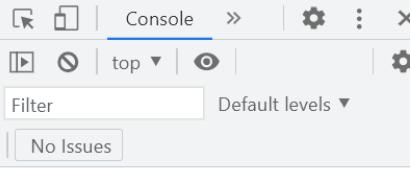
// като закачим за прозореца, и може да си тестваме функциите от data.js в конзолата на браузъра

```
import * as api from './api/api.js';
```

```
// window.api = api;
```

```
< → C 127.0.0.1:5500
```

Bookmarks Dict Online platforms



```
/*debug
```

```
import * as api from './api/data.js';
```

```
window.api = api;
```

```
*/
```

//Контекстът се препредава като започнем от decorateContext, и във всяка следваща функция през page.js

```
page(decorateContext); //ще се изпълни винаги /ще се закачи винаги //глобален middle-ware
```

//вместо да викаме всеки път decorateContext функцията като chain-ване

```
// page('/home', decorateContext, catalogPage); //Chain-ваме route handlers - така наречения middle-ware pattern, то от Page.js са измислили да се декларира в началото и без първи параметър
```

```
page('/home', catalogPage);
```

```
page('/details/:id', loadMovie, detailsPage); // middle-ware pattern
```

```
page('/login', loginPage);
```

```
page('/register', registerPage);
```

```
page('/create', createPage);
```

```
page('/edit/:id', loadMovie, editPage); // middle-ware pattern
```

```
page('/', '/home');
```

```
updateUserNav();
```

```
page.start(); //Закачане на eventListener-ите
```

```
//next is callback
```

//цялата тази функция decorateContext се нарича middle-ware

```
async function decorateContext(ctx, next) {
```

```
 ctx.renderProp = (template) => render(template, root); //задай пропърти renderProp
```

```
 ctx.updateUserNav = updateUserNav; //задай пропърти updateUserNav за прилагане в други scope-ове
```

```
 next(); //аз приключих, всичко е наред, можеш да извикаш следващия
```

```

}

function updateUserNav() {
 const userData = getUserData();
 if (userData) {
 [...document.querySelectorAll('nav .user')].forEach(e => e.style.display = 'inline-block');
 [...document.querySelectorAll('nav .guest')].forEach(e => e.style.display = 'none');
 document.getElementById('welcomeMsg').textContent = `Welcome, ${userData.email}`;
 //се изписва като логнат потребител
 } else {
 [...document.querySelectorAll('nav .user')].forEach(e => e.style.display = 'none');
 [...document.querySelectorAll('nav .guest')].forEach(e => e.style.display = 'inline-block');
 }
}

```

//няма нужда тук да викаме preventDefault на event-а, защото в html-а имаме

//

//тази функция можем да я извикаме и с addEventListener

```

async function onLogout() {
 await logout();
 updateUserNav(); // в същия scope
 page.redirect('/');
}

```

**При logout, по-добре да не използваме тук async/await**

//нас не ни интересува какво ще отговори сървъра, важното е да изчистим при нас сесията.

Затова не правим async

//защото сървъра ако даде грешка, няма да се изпълни updateUserNav

```

function onLogout() {
 logout();
 updateUserNav(); // в същия scope
 page.redirect('/');
}

```

*login.js*

```

import {html} from '../lib.js';
import {login} from '../api/data.js';

const loginTemplate = (onSubmit) => html`

<section id="form-login">

 <form @submit=${onSubmit} class="text-center border border-light p-5" action=""

method="">>

 <div class="form-group">

 <label for="email">Email</label>

```

```

 <input type="text" class="form-control" placeholder="Email" name="email"
value="">
 </div>
 <div class="form-group">
 <label for="password">Password</label>
 <input type="password" class="form-control" placeholder="Password"
name="password"
 value="">
 </div>

 <button type="submit" class="btn btn-primary">Login</button>
 </form>
</section>
`;

export function loginPage(ctx) {
 ctx.renderProp(loginTemplate(onSubmit));

 //event listener-а няма опасност да бъде добавен два пъти
 async function onSubmit(event) {
 event.preventDefault();
 const formData = new FormData(event.target);

 const email = formData.get('email').trim();
 const password = formData.get('password').trim();

 await login(email, password);
 ctx.updateUserNav();
 ctx.page.redirect('/home'); //самият контекст носи в себе си page.js библиотеката, от
която можем да редиректнем.
 }
}

```

```

register.js
import { register } from "../api/data.js";
import { html } from "../lib.js";

const registerTemplate = (onSubmit) => html`

<div class="col-md-12">
 <h1>Register New User</h1>
 <p>Please fill all fields.</p>
 </div>
</div>
<form @submit=${onSubmit}>
 <div class="row space-top">
 <div class="col-md-4">
 <div class="form-group">
 <label class="form-control-label" for="email">Email</label>


```

```

 <input class="form-control" id="email" type="text" name="email">
 </div>
 <div class="form-group">
 <label class="form-control-label" for="password">Password</label>
 <input class="form-control" id="password" type="password" name="password">
 </div>
 <div class="form-group">
 <label class="form-control-label" for="rePass">Repeat</label>
 <input class="form-control" id="rePass" type="password" name="rePass">
 </div>
 <input type="submit" class="btn btn-primary" value="Register" />
</div>
</div>
</form>`;

export function registerPage(ctx) {
 ctx.renderProp(registerTemplate(onSubmit));

 //event listener-а няма опасност да бъде добавен два пъти
 async function onSubmit(event) {
 event.preventDefault();
 const formData = new FormData(event.target);

 const email = formData.get('email').trim();
 const password = formData.get('password').trim();
 const rePass = formData.get('rePass').trim();

 if (email == '' || password == '') {
 return alert('All fields are required!');
 }

 if (password != rePass) {
 return alert('Passwords don\'t match!');
 }

 await register(email, password);
 // ctx.updateUserNav(); тук няма нужда все още да сменяме пач бутоните
 ctx.page.redirect('/');
 }
}

```

*create.js*

```

import {html} from '../lib.js';

const createTemplate = () => html`

<section id="add-movie">

<form class="text-center border border-light p-5" action="#" method="">

 <h1>Add Movie</h1>

```

```

<div class="form-group">
 <label for="title">Movie Title</label>
 <input type="text" class="form-control" placeholder="Title" name="title"
value="">
</div>
<div class="form-group">
 <label for="description">Movie Description</label>
 <textarea class="form-control" placeholder="Description"
name="description"></textarea>
</div>
<div class="form-group">
 <label for="imageUrl">Image url</label>
 <input type="text" class="form-control" placeholder="Image Url" name="imageUrl"
value="">
</div>
 <button type="submit" class="btn btn-primary">Submit</button>
</form>
</section>
`;

export function createPage(ctx) {
 ctx.renderProp(createTemplate());
}

```

*catalog.js*

```

import { getAllMovies } from '../api/data.js';
import {html, until} from '../lib.js';

// Първо си правим един темплейт статичен, и след това изкарваме от него втори темплейт за
// всеки филм. При което става динамичен templating.
const catalogTemplate = (moviePromise) => html`

<section id="home-page">

 <div class="jumbotron jumbotron-fluid text-light" style="background-color: #343a40;">

 <h1 class="display-4">Movies</h1>
 <p class="lead">Unlimited movies, TV shows, and more. Watch anywhere. Cancel

anytime.</p>
 </div>

 <h1 class="text-center">Movies</h1>

 <section id="add-movie-button">

 Add Movie
 </section>

 <section id="movie">
 <div class=" mt-3 ">

```

```

<div class="row d-flex d-wrap">

 <div class="card-deck d-flex justify-content-center">
 ${until(moviePromise, html`<p>Loading… </p>`)}
 </div>
</div>
</div>
</section>

</section>`;

const movieCard = (movie) => html`

<div class="card mb-4">

 <img class="card-img-top" src=${movie.img}

 alt="Card image cap" width="400">

 <div class="card-body">

 <h4 class="card-title">${movie.title}</h4>

 </div>

 <div class="card-footer">

 <button type="button" class="btn btn-info">Details</button>

 </div>
</div>`;

```

```

export function catalogPage(ctx) {
 ctx.renderProp(catalogTemplate(loadMovies()));
}

async function loadMovies() {
 const movies = await getAllMovies(); //масив от филми

 return movies.map(movieCard); //масив от рендирани темплейти за всеки филм
}

```

*home.js (от друга задача)*

```

import {html} from '../lib.js';
import { getUserData } from '../util.js';

const homeTemplate = () => html`

<section id="welcome">

 <div id="welcome-container">

 <h1>Welcome To Meme Lounge</h1>

 <h2>Login to see our memes right away!</h2>

 <div id="button-div">

 Login

 Register
 </div>
 </div>
</section>`;

```

```

 </div>
</section>`;

export function HomePage(ctx){
 //слагане на guard, по принцип се слагало в middleware – ако сме логнати, и заредим
 //начална страница, то да ни redirect-не към /memes страницата
 if (getUserData()) {
 ctx.page.redirect('memes');
 } else {
 ctx.renderProps(homeTemplate());
 }
}

```

```

details.js

import { getMovieById } from '../api/data.js';
import {html, until} from '../lib.js';
import { getUserData } from '../util.js';

const detailsTemplate = (moviePromise) => html`

<section id="movie-details">

 ${until(moviePromise, html`<p>Loading …</p>`)}

</section>`;

const movieTemplate = (movie) => html`

<div class="container">

 <div class="row bg-light text-dark">

 <h1>Movie title: ${movie.title}</h1>

 <div class="col-md-8">

 alt="Movie">

 </div>

 <div class="col-md-4 text-center">

 <h3 class="my-3 ">Movie Description</h3>

 <p>${movie.description}</p>

 ${movie.isOwner

 ? html`

 Delete

 Edit

 : html`

 Like`}

 Liked 1

 </div>

 </div>

</div>`;

export function detailsPage(ctx) {
 const movieId = ctx.params.id;

```

→ C 127.0.0.1:5500/details/34a1cab1-81f1-47e5-aec3-ab6c9810efe1

```
ctx.renderProp(detailsTemplate(loadMovie(ctx)));
}

async function loadMovie(ctx) {
 const movie = await ctx.moviePromise; //тук го await-ваме реално

 const userData = getUserData();
 const = userData && userData.id == movie._ownerId;

 if () {
 movie.isOwner = true; //задаваме ново свойство на филма
 }

 return movieTemplate(movie);
}
```

edit.js

```
import {html, until} from '../lib.js';

const editTemplate = (moviePromise) => html`

<section id="edit-movie">

 ${until(moviePromise, html`

Loading…

`)}

</section>`;

 <textarea name="description" id="description" .value=${book.description}></textarea>

 input → value

 <input type="text" name="imageUrl" id="image" .value=${book.imageUrl}>

 textarea → value

 <select id="type" name="type" value="Fiction">

 <option value="Fiction" selected>Fiction</option>

 <option value="Romance">Romance</option>

 <option value="Mystery">Mystery</option>

</select>

 select → value

 value

 значаи се низът на данни

textContent (innerHTML)
`;
```

```
const formTemplate = (movie, onSubmit) => html`

<form class="text-center border border-light p-5" action="#" method="">

 <h1>Edit Movie</h1>

 <div class="form-group">

 <label for="title">Movie Title</label>
```

```

<input type="text" class="form-control" placeholder="Movie Title" .value=${movie.title}
//задай му нова стойност през lit-html като го слага като свойство с точката. Ако е без
точка пак може, но тогава е атрибут. За textarea задължително .value
 name="title">
</div>
<div class="form-group">
 <label for="description">Movie Description</label>
 <textarea class="form-control" placeholder="Movie Description..." name="description"
.value=${movie.description}></textarea> //задай му нова стойност през lit-html като го слага
како свойство с точката. Ако е без точка пак може, но тогава е атрибут. За textarea
задължително .value
</div>
<div class="form-group">
 <label for="imageUrl">Image url</label>
 <input type="text" class="form-control" placeholder="Image Url" .value=${movie.img}
 name="imageUrl">
</div>
 <button type="submit" class="btn btn-primary">Submit</button>
</form>`;

export function editPage(ctx) {
 ctx.renderProp(editTemplate(loadMovie(ctx)));
}

async function loadMovie(ctx) {
 const movie = await ctx.moviePromise;

 return formTemplate(movie, onSubmit);

 async function onSubmit(event) {
}
}

```

*Когу се resolve-не promise-a*

```

import { html, until } from '../lib.js';
import { getAllPosts } from '../api/data.js';

//Първо си правим един темплейт статичен, и след това изкарваме от него втори темплейт за
всеки филм. При което става динамичен templating.
const dashboardFullTemplate = (postsPromise) => html`

<section id="dashboard-page">
 <h1 class="title">All Posts</h1>

 <div class="all-posts">
 ${until(postsPromise, html`<p>Loading… </p>`)}
 </div>
</section>`;

const dashCard = (aPost) => html`


```

```

<div class="post">
 <h2 class="post-title">${aPost.title}</h2>

 <div class="btn-wrapper">
 Details
 </div>
</div>`;

export function dashboardPage(ctx) {
 ctx.renderProp(dashboardFullTemplate(loadPosts()));
}

async function loadPosts() {
 const dashPosts = await getAllPosts(); //масив от постове, resolve-нат!!!
 if (dashPosts.length == 0) {
 return html`<h1 class="title no-posts-title">No posts yet!</h1>`;
 } else {
 return dashPosts.map(dashCard); //масив от рендирани темплейти за всеки пост
 }
}

```

Дебъгване на formData

```

async function onSubmit(event) {
 event.preventDefault();
 const formData = new FormData(event.target);
 //debug
 console.log(...formData.entries());

```

```

 ▼ Array ["title", "My title"]
 0: "title"
 1: "My title"
 length: 2
 ▶ <prototype>: Array []
 ▼ Array ["description", "ala bala
description"]
 0: "description"
 1: "ala bala description"
 length: 2
 ▶ <prototype>: Array []
 ▼ Array ["imageUrl", "rrrr"]
 0: "imageUrl"
 1: "rrrr"
 length: 2
 ▶ <prototype>: Array []
 ▼ Array ["type", "Fiction"]
 0: "type"
 1: "Fiction"
 length: 2
 ▶ <prototype>: Array []

```

```

//debug
 console.log(...formData.values());
myTitle My short description image here yea ↴
Fiction

```

---

```

const title = formData.get('title').trim();
const description = formData.get('description').trim();
const imageUrl = formData.get('imageUrl').trim();
const type = formData.get('type').trim();

```

```

formData.values().some(v => v == ''); //при повече полета за по-кратък запис
if (title == '' || description == '' || imageUrl == '') {
 return alert('Please, fill in all fields');
}

```

Изнесен тернарен оператор и реализиране на Likes – пример:

```

detailsBook

import { deleteBook, getBookById, getLikesByBookId, getMyLike10r0ByBookId, likeBook } from
'../api/data.js';
import { html } from '../lib.js';
import { getUserData } from '../util.js';

const detailsBookTemplate = (book, isOwner, onDelete, likes, showLikeButton, onLike) => html`

<!-- Details Page (for Guests and Users) -->
<section id="details-page" class="details">
 <div class="book-information">
 <h3>${book.title}</h3>
 <p class="type">Type: ${book.type}</p>
 <p class="img"></p>
 <div class="actions">
 <!-- Edit/Delete buttons (Only for creator of this book) -->
 ${bookControlsTemplate(book, isOwner, onDelete)}
 </div>
 </div>
 <!-- Like button (Only for logged-in users, who are not creators of the current book) -->
 ${likeControlsTemplate(showLikeButton, onLike)}

 <!-- (for Guests and Users) -->
 <div class="likes">

 Likes: ${likes}
 </div>
</div>
<div class="book-description">
 <h3>Description:</h3>
 <p>${book.description}</p>
</div>
</section>`;

const bookControlsTemplate = (book, isOwner, onDelete) => {
 if (isOwner) {
 return html`

 Edit
 <a @click=${onDelete} class="button" href="javascript:void(0)">Delete`;
 } else {
 return null;
 }
}

```

```

const likeControlsTemplate = (showLikeButton, onLike) => {
 if (showLikeButton) {
 return html``;
 } else {
 return null;
 }
}

export async function detailsBookPage\(ctx\) {
 const userData = getUserData\(\);

 const \[book, likes, hasLike\] = await Promise.all\(\[
 getBookById\(ctx.params.id\),
 getLikesByBookId\(ctx.params.id\),
 userData ? getMyLike1Or0ByBookId\(ctx.params.id, userData.id\) : 0
 \]\);

 const isOwner = userData && userData.id == book._ownerId;
 const showLikeButton = userData != null && isOwner == false && hasLike == false;

 ctx.renderProps\(detailsBookTemplate\(book, isOwner, onDelete, likes, showLikeButton, onLike\)\);

 async function onDelete\(\) {
 const choice = confirm\(`Are you sure you want to delete \${book.title}`\);
 if \(choice\) {
 await deleteBook\(ctx.params.id\);
 ctx.page.redirect\('/'\);
 }
 }

 async function onLike\(\) {
 console.log\('Liked book'\);
 await likeBook\(ctx.params.id\);
 ctx.page.redirect\('/details/' + ctx.params.id\);
 }
}

data.js
export async function getLikesByBookId\(bookId\) {
 return api.get\(`/data/likes?where=bookId%3D\${bookId}%22&distinct=_ownerId&count`\);
}

export async function getMyLike1Or0ByBookId\(bookId, userId\) {
 return
 api.get\(`/data/likes?where=bookId%3D\${bookId}%22%20and%20_ownerId%3D\${userId}%22&count`\);
}

//debug

```

```
// window.likeBook = likeBook;
// window.getLikesByBookId = getLikesByBookId;
// window.getMyLike1Or0ByBookId = getMyLike1Or0ByBookId;
```

Search обикновен - пример:

```
import { searchBooks } from '../api/data.js';
import {html} from '../lib.js';
import { bookCard } from './common.js';

const searchTemplate = (books, onSearch, params = '') => html`

<!-- Dashboard Page (for Guests and Users) -->

<section id="search-page" class="dashboard">

 <h1>Search</h1>
 <form @submit=${onSearch}>
 <input type="text" name="search" .value=${params}>
 <input type="submit" value="Search">
 </form>

 ${books.length == 0
 ? html`<p class="no-books">No results found/p>`
 : html`<ul class="other-books-list">${books.map(bookCard)}`}
 }
</section>`;

export async function searchPage(ctx) {
 const params = ctx.querystring.split('=')[1]; да, хващаме го от URL-а ⓘ
 let books = [];

 if (params) {
 books = await searchBooks(decodeURIComponent(params));
 }

 ctx.renderProps(searchTemplate(books, onSearch, params));

 function onSearch(event) {
 event.preventDefault();
 const formData = new FormData(event.target);
 const search = formData.get('search');

 if (search != '') {
 ctx.page.redirect('search?query=' + encodeURIComponent(search)); да,
 слагаме го в URL-а ⓘ
 console.log(search);
 }
 }
}
```

Search advanced - пример:

querystringMiddleware.js

Реално тук хваща от URI: {search: alabala, page: alaBala, ....}

```

export function querystringMiddleware(ctx, next) {
 let qs = {};

 if (ctx.querystring) {
 qs = ctx.querystring
 .split('&')
 .map(x => x.split('='))
 .reduce((a, [key, value]) => {
 a[key] = value;
 return a;
 }, {});
 }

 ctx.qs = qs;

 next();
}

```

movieService.js

```

export const search = (text) =>
request.get(`${apiEndPnts.movies}?where=title%20LIKE%20"${text}"`); //we returns a promise
from the request.get()

```

app.js

```

page(querystringMiddleware); //
```

//Генералните route-ве да са по-надолу изброени, а специфичните route-ве по-нагоре изброени

```

page('/', HomePage);
page('/login', LoginPage);
page('/movies', moviesAllPage);
page('/movies/add', addMoviePage); //!!! така не се бие със следващия route /movies/:movieId
page('/my-movies', myMoviesPage);
page('/movies/:movieId/edit', editMoviePage); //!!! така не се бие със следващия route
/movies/:movieId
page('/movies/:movieId', movieDetailsPage);
page('/movies/:movieId/delete', deleteMoviePage);
```

moviesView.js

```

export function moviesAllPage(ctx) {
 let moviesPromise = null;

 console.log(ctx.qs);
 if (ctx.qs.search) {
 moviesPromise = movieService.search(ctx.qs.search);
 } else if (ctx.qs.page) {
 //some other feature
 } else {
```

```

 moviesPromise = movieService.getAll();
 }

 moviesPromise
 .then(movies => ctx.renderProp(moviesAllTemplate(movies))); // here in the then the
promise is resolved
}

export function myMoviesPage(ctx) {
 movieService.getMyMovies(ctx.userId)
 .then(movies => {
 // console.log(movies);
 ctx.renderProp(moviesAllTemplate(movies)); // here in the then the promise is
resolved
 });
}

```

28.0. Следващите точки от 27ма лекция реално са в папката на 28-ма лекция  
27.7. При непопълнени данни/mandatory fields полета за грешка се появяват - production изпълнение с наченки на React

```

edit.js
import { editItem, getById } from "../api/data.js";
import { html, until } from "../lib.js";

const editTemplate = (itemPromise) => html`

Edit Furniture

Please fill all fields.

${until(itemPromise, html`

Loading …

`)
</div>};

const formTemplate = (item, onSubmit, errorMsg, errors) => html` ${errorMsg}

` : null};//изписване
на грешката на страницата

Make

`


```

```

 <div class="form-group has-success">
 <label class="form-control-label" for="new-model">Model</label>
 <input class={'form-control' + (errors.model ? ' is-invalid' : '')} id="new-model" type="text" name="model" .value=${item.model}>
 </div>
 <div class="form-group has-danger">
 <label class="form-control-label" for="new-year">Year</label>
 <input class={'form-control' + (errors.year ? ' is-invalid' : '')} id="new-year" type="number" name="year" .value=${item.year}>
 </div>
 <div class="form-group">
 <label class="form-control-label" for="new-description">Description</label>
 <input class={'form-control' + (errors.description ? ' is-invalid' : '')} id="new-description" type="text" name="description" .value=${item.description}>
 </div>
 </div>
 <div class="col-md-4">
 <div class="form-group">
 <label class="form-control-label" for="new-price">Price</label>
 <input class={'form-control' + (errors.price ? ' is-invalid' : '')} id="new-price" type="number" name="price" .value=${item.price}>
 </div>
 <div class="form-group">
 <label class="form-control-label" for="new-image">Image</label>
 <input class={'form-control' + (errors.img ? ' is-invalid' : '')} id="new-image" type="text" name="img" .value=${item.img}>
 </div>
 <div class="form-group">
 <label class="form-control-label" for="new-material">Material (optional)</label>
 <input class="form-control" id="new-material" type="text" name="material" .value=${item.material}>
 </div>
 <input type="submit" class="btn btn-info" value="Edit" />
 </div>
</div>
`;

export function editPage(ctx) {
 const itemPromise = getById(ctx.params.id);

 update(itemPromise, null, {});

 function update(itemPromise, errorMsg, errors) {
 ctx.renderProp(editTemplate(loadItem(itemPromise, errorMsg, errors)));
 }

 async function loadItem(itemPromise, errorMsg, errors) {
 const item = await itemPromise;

```

```

 return formTemplate(item, onSubmit, errorMsg, errors);
 }

 async function onSubmit(event) {
 event.preventDefault();

 const formData = [...(new FormData(event.target)).entries()];
 const data = formData.reduce((a, [k, v]) => Object.assign(a, {[k]: v.trim()}), {});

 //списък от полетата, които са празни. Като материала не е задължителен и може да е
 //празен
 const missing = formData.filter(([inputKey, v]) => inputKey != 'material' && v.trim() == '');
 console.log(missing);

 try {
 if (missing.length > 0) {
 const errors = missing.reduce((a, [k]) => Object.assign(a, {[k] : true})), {};
 //непопълнените задължителни полета ги направи на обект от ключ и стойност
 console.log(errors);
 throw {
 error: new Error('Please fill in all mandatory fields!'),
 errors
 }
 // return alert('Please, fill all mandatory fields');
 }
 }
 }

 data.year = Number(data.year);
 data.price = Number(data.price);

 const result = await editItem(ctx.params.id, data);
 event.target.reset();
 ctx.page.redirect('/details/' + result._id);
} catch (err) {
 const message = err.message || err.error.message; //Хваща първо грешка със
 //сървъра, и чак след това хваща грешка свързана с user наша си
 update(data, message, err.errors || {});
}
}
}
}

```

## 27.8. Queries с Pagination и Search, както и за Notification и Modal

*data.js*

```

import * as api from './api.js';

export const login = api.login;
export const register = api.register;
export const logout = api.logout;

const pageSize = 4;

```

```

const endpoints = {
 all: '/data/catalog?pageSize=4&offset=',
 count: '/data/catalog?count', //идва от сървъра
 getById: '/data/catalog/',
 myItems: (userId) => `/data/catalog?where=_ownerId%3D${userId}%22`,
 countMyItems: (userId) => `/data/catalog?where=_ownerId%3D${userId}%22&count`,
 create: '/data/catalog',
 edit: '/data/catalog/',
 delete: '/data/catalog/'
}

export async function getAll(page, search) {
 let urlGetItems = endpoints.all + (page - 1) * pageSize;
 let urlCountItems = endpoints.count;

 //pagination и search работи - сървъра го прави/връща по правилния начин
 if (search) { //скипва празния стринг
 urlGetItems += '&where=' + encodeURIComponent(`make LIKE "${search}"`); //сървъра в
 случай търси с &where=attribute LIKE "123"
 urlCountItems += '&where=' + encodeURIComponent(`make LIKE "${search}"`);
 }

 const [data, count] = await Promise.all([
 api.get(urlGetItems),
 api.get(urlCountItems),
]);

 return {
 data,
 totalPages: Math.ceil(count / pageSize)
 }
}

export async function getMyItems(userId, page, search) {
 let urlGetMyItems = endpoints.all + (page - 1) * pageSize;
 let urlCountMyItems = endpoints.count;

 //pagination и search работи - сървъра го прави/връща по правилния начин
 if (search) { //скипва празния стринг
 urlGetMyItems += '&where=' + encodeURIComponent(`make LIKE "${search}"`); //сървъра в
 случай търси с &where=attribute LIKE "123"
 urlCountMyItems += '&where=' + encodeURIComponent(`make LIKE "${search}"`);
 }

 const [data, count] = await Promise.all([
 api.get(urlGetMyItems),
 api.get(urlCountMyItems),
]);
}

```

```

 return {
 data: data,
 totalPages: Math.ceil(count / pageSize)
 }
}

catalog.js
import { html, until } from "../lib.js";
import { getAll, getMyItems } from "../api/data.js";
import { getUserData, parseQueryString } from "../util.js";

const catalogTemplate = (dataPromise, userpage, onSearch, search) => html`

<div class="col-md-12">
 ${userpage
 ? html`<h1>My Furniture</h1>
 <p>This is a list of your publications.</p>`
 : html`<h1>Welcome to Furniture System</h1>
 <p>Select furniture from the catalog to view details.</p>`}
 </div>
 <div class="col-md-12">
 <form @submit=${onSearch}>
 <input type="text" name="search" .value=${search}>
 <input type="submit" value="Search">
 </form>
 </div>
</div>
<div class="row space-top">
 <!--
 1
 2
 3-->
 ${until(dataPromise, html`<p>Loading... </p>`)}
</div>`;

// ${currPage > 1 ? html` < Prev` : null}
// ${currPage < totalPages ? html`Next >` : null}

const pagerTemplate = (currPage, totalPages, search) => html`

${currPage > 1 ? html` < Prev` : null}
 ${currPage < totalPages ? html`Next >` : null}
</div>`;


```

```

const itemTemplate = (item) => html`

<div class="col-md-4">

 <div class="card text-white bg-primary">

 <div class="card-body">

 <p>${item.description}</p>

 <footer>

 <p>Price: ${item.price} $</p>

 </footer>

 <div>

 Details

 </div>

 </div>

 </div>

</div>`;

function createPageHref(currPage, step, search) {

 return `?page=${currPage + step}` + (search ? `&search=${search}` : '');

}

export function catalogPage(ctx) {

 console.log(ctx);

 const query = parseQueryString(ctx.querystring);

 console.log("query", query);

 const page = Number(query.page || 1);

 console.log("s", query.search);

 const search = query.search || '';

 const userpage = ctx.pathname == '/my-furniture';

 ctx.renderProp(catalogTemplate(loadItems(userpage, page, search), userpage, onSearch,

 search));

 function onSearch(event) {

 event.preventDefault();

 const formData = new FormData(event.target);

 const searchParam = formData.get('search').trim();

 if (searchParam) { //ако има searchParam

 console.log("ss", search);

 ctx.page.redirect(`?search=${searchParam}`);

 } else {

 ctx.page.redirect(`/`);

 }

 // ctx.page.redirect(`?search=${searchParam}`);

 }
}

```

```

async function loadItems(userpage, page, search) {
 let items = [];
 if (userpage) {
 const userId = getUserData().id;
 items = await getMyItems(userId, page, search); //тук не бачка все още
 } else {
 items = await getAll(page, search); //масив от записи от сървъра
 }

 return [
 pagerTemplate(page, items.totalPages, search),
 ...items.data.map(itemTemplate) //масив от рендирани темплейти за всеки item ги
 правим на поредица от данни заедно за html-а да е един цял
];
}

```

*utils.js*

```

export function parseQueryString(string) {
 'page=3&search=chair';
 const params = string
 .split('&')
 .map(p => p.split('='))
 .reduce((a, [currKey, currValue]) => Object.assign(a, {[currKey]: currValue}), {});

 return params;
}

```

## 27.9. Notification – обикновено в горната дясна част на екрана

*index.html*

```

<!-- <div id="notification">
 Hello
 ✖
</div>-->

```

*style.css*

```

#notification {
 position: fixed;
 top: 50px;
 right: 300px;
 background-color: #f84a4a;
 color: black;
 padding: 16px 32px;
 cursor: pointer;
}

```

```
✓ src
 ✓ api
 JS api.js
 JS data.js
 ✓ common
 JS notify.js
```

*notify.js*

```
const element = document.createElement('div');
element.id = 'notification';
const msg = document.createElement('span');
const closeBtn = document.createElement('span');
closeBtn.innerHTML = 'Ϯ';
element.appendChild(msg);
element.appendChild(closeBtn);

element.addEventListener('click', clear);

export function notify(message) {
 msg.textContent = message;
 document.body.appendChild(element);

 setTimeout(clear, 4000);
}
```

```
export function clear() {
 element.remove();
}
```

*notify.js - От друга задача*

```
<!-- Notifications -->
<section id="notifications">
 <div id="errorBox" class="notification">
 MESSAGE
 </div>
</section>

const element = document.getElementById('errorBox');
const output = element.querySelector('span');

export function notify(msg) {
 output.textContent = msg;
 element.style.display = 'block';
 setTimeout(() => element.style.display = 'none', 3000);
}
```

*api.js*

```
import {notify} from '../common/notify.js';
```

```

catch (err) {
 // alert(err.message);
 notify(err.message);
 throw err;
}

```

## 27.10. Modal dialog box – спира изпълнението на браузъра

За разлика от обикновения стандартен по подразбиране диалогов прозорец на дадения браузър, където имаме достъп до страницата и преди потребителят да е затворил диалоговия прозорец.

При modal dialog box имаме един елемент overlay, който скрива/покрива всички останали елементи/всичко останало. И така не може да се цъка по страницата.

1. Потребителят не може да гласува да не получава повече модали. Потребителят не може да го изключи. А при confirmation dialog box на браузъра може да спрат да се показват.
2. Стилизацията/по-красив
3. В един модал можем да си сложим каквите искаме бутони в него.

*index.html*

```

<!--<div id="overlay">
 <div id="modal">
 <p>Are you sure</p>
 <div>
 <button>Ok</button>
 <button>Cancel</button>
 </div>
 </div>
</div>-->

```

*style.css*

```

#overlay {
 background-color: rgba(128, 128, 128, .6);
 position: fixed;
 top: 0;
 left: 0;
 right: 0;
 bottom: 0;
}

#modal {
 background-color: white;
 width: 350px;
 padding: 16px 32px;
 text-align: center;
 color: black;
 margin: auto;
 margin-top: 15vh;
}

```

```
}
```

```
details.js
```

```
export function detailsPage(ctx) {
 ctx.renderProp(detailsTemplate(loadItem(ctx.params.id, onDelete)));

 async function onDelete() {
 const choice = showModal('Are you sure you want to delete this item?');

 if (choice) {
 await deleteItem(ctx.params.id);
 ctx.page.redirect('/');
 }

 /*
 const choice = confirm('Are you sure you want to delete this item?');
 if (choice) {
 await deleteItem(ctx.params.id);
 ctx.page.redirect('/');
 }*/
 }
}
```

```
modal.js
```

```
const element = document.createElement('div');
element.id = 'overlay';

element.innerHTML = `
<div id="modal">
 <p>Are you sure</p>
 <div>
 <button class="modal-ok">Ok</button>
 <button class="modal-cancel">Cancel</button>
 </div>
</div>`;

element.querySelector('.modal-ok').addEventListener('click', () => onChoice(true));
element.querySelector('.modal-cancel').addEventListener('click', () => onChoice(false));

const msg = element.querySelector('p');

let onChoice = null;

export function showModal(message) {
 msg.textContent = message;
 document.body.appendChild(element);

 //връщане на резултат с Promise
 return new Promise(resolveCallback => {
```

```

 onChoice = (choice) => {
 clear();
 resolveCallback(choice);
 }
 });

export function clear() {
 element.remove();
}

```

## 27.11. Alert

```

catch (err) {

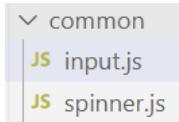
 alert(err.message);
 throw err;
}

```

## 27.12. Reveal password

Трябва да сменим type от type="password" на type="text"

27.13. (common folder) - classMap използване от lit-html + изнасяне някои функционалности на onSubmit функцията в createSubmitHandler



```


import { html, classMap } from "../lib.js";

export const input = (label, type, name, value = '', hasError) => {
 if (type == 'textarea') {
 return html`${label}</label>
<${type} name=${name} .value=${value}><${type}></${type}>`;
 } else {
 return html`${label}</label><${type} type=${type} name=${name}
.value=${value}></${type}>`;
 }
};

/*
<label>Email<${type} type="text" name="email" .value=${values.email}></label>
<label>Display name<${type} type="text" name="username" .value=${values.username}></label>*/

```

```
spinner.js
import {html} from '../lib.js';

export const spinner = () => html`<p class="spinner">Loading …</p>`;

register.js
import { register } from '../api/data.js';
import { input } from '../common/input.js';
import { html } from '../lib.js';
import { createSubmitHandler } from '../util.js';

const registerTemplate = (onSubmit, errorMsg, errors, values) => html`

<header>
 <h1>Register</h1>
</header>
<form @submit=${onSubmit}>
 ${errorMsg ? html`<p class="error-msg">${errorMsg}</p>` : null}
 //по-добър начин при повече елементи:
 ${input('Email', 'text', 'email', values.email, errors.email)}
 ${input('Display name', 'text', 'username', values.username, errors.username)}
 ${input('Password', 'password', 'password', values.password, errors.password)}
 ${input('Repeat Pass', 'password', 'repass', values.repass, errors.repass)}
 <input class="action" type="submit" value="Sign Up">
</form>
</div>`;

export function registerPage(ctx) {
 update();

 function update(errorMsg, errors = {}, values = {}) {
 ctx.renderProp(registerTemplate(createSubmitHandler(onSubmit, 'email', 'username',
'password', 'repass'),
 errorMsg, errors, values));
 }

 async function onSubmit(data, event) {
 try {
 const missing = Object.entries(data).filter(([k, v]) => v == '');
 if (missing.length > 0) {
 const errors = missing.reduce((a, [k]) => Object.assign(a, { [k]: true }), {});
 throw {
 error: new Error('All fields are required!'),
 errors
 };
 }
 } catch (err) {
 console.error(err);
 }
 }
}


```

```

 }
 }

 if (data.password != data.repass) {
 throw {
 error: new Error('Passwords don\'t match!'),
 errors: {
 password: true,
 repass: true
 }
 }
 }

 await register(data.email, data.username, data.password);
 event.target.reset();
 ctx.updateUserNavProp();
 ctx.page.redirect('/topics');

} catch (err) { //грешките, които идват от сървъра нямат поле error, а имат поле
message
 const message = err.message || err.error.message;
 update(message, err.errors, data);
}
}
}
}

```

*util.js*

```

export function createSubmitHandler(callback, ...fieldNames) {
 return function (event) {
 event.preventDefault();
 const formData = new FormData(event.target);

 const resultData = {};

 for (let field of fieldNames){
 console.log(field, ': ', formData.get(field));
 resultData[field] = formData.get(field).trim();
 }

 callback(resultData, event); //подаваме и event-а, защото искаме да reset-нем в
onSibmit-а на register.js
 };
}

```

*create.js*

```

import { createTopic } from '../api/data.js';
import { input } from '../common/input.js';
import { html } from '../lib.js';

```

```

import { createSubmitHandler } from './util.js';

const createTemplate = (onSubmit, errorMsg, errors, values) => html`

<header>
 <h1>Create New Topic</h1>
</header>
<form @submit=${onSubmit}>
 ${errorMsg ? html`<p class="error-msg">${errorMsg}</p>` : null}
 ${input('Topic title', 'text', 'title', values.title, errors.title)}
 ${input('Content', 'textarea', 'content', values.content, errors.content)}
 <div class="center">
 <input class="action" type="submit" value="Publish topic">
 </div>
</form>
</div>`;

export function createPage(ctx) {
 update();

 function update(errorMsg = '', errors = {}, values = {}) {
 ctx.renderProp(createTemplate(createSubmitHandler(onSubmit, 'title', 'content'),
 errorMsg,
 errors,
 values));
 }

 async function onSubmit(data) {
 try {
 const missing = Object.entries(data).filter(([k, v]) => v === '');
 console.log(missing);
 if (missing.length > 0) {
 const errors = missing.reduce((a, [k]) => Object.assign(a, { [k]: true }), {});
 throw {
 error: new Error('All fields are required!'),
 errors
 }
 }
 }

 //redirect-ваме към детайлите на новосъздадената тема
 const result = await createTopic(data);
 console.log(result);
 ctx.page.redirect('/topic/' + result._id);
 } catch (err) {
 const message = err.message || err.error.message;
 update(message, err.errors, data);
 }
}
}


```

```
details.js

import { getTopicById, getCommentsByTopicId, createComment} from '../api/data.js';
import { spinner } from '../common/spinner.js';
import {html, until} from '../lib.js';
import { createSubmitHandler, getUserData } from '../util.js';

const detailsATopicTemplate = (topicPromise) => html`

${until(topicPromise, spinner())}

`;

const topicCard = (topic, isOwner, comments, onCommentSubmit) => html`

${topic.title}

${isOwner
? html`Edit <a class="action"
href="javascript:void(0)">Delete
: html`Topic created by ${topic.author.username}`}

${topic.content}

${commentForm(onCommentSubmit)}
${comments.map(commentCard)}

`;

const commentCard = (comment) => html`

By ${comment.author.username}

${comment.content}

`;

const commentForm = (onCommentSubmit) => html`

Post new comment

<form @submit=${onCommentSubmit}>
 <label>Content <textarea name="content"></textarea></label>
 <input class="action" type="submit" value="Post">
</form>

`;
```

```

let _topicData = null;

export function detailsATopicPage(ctx) {
 _topicData = getTopicById(ctx.params.id);
 update();

 function update() {
 ctx.renderProp(detailsATopicTemplate(loadTopic(ctx.params.id, onCommentSubmit)));
 }

 async function onCommentSubmit(data, event) {
 if (data.content == '') {
 return alert('Cannot post empty comment!')
 }

 //с цел да не цъкнем два пъти на един и същи бутон/да не се получи да изпратим
 //формата два пъти
 [...event.target.querySelectorAll('input', 'textarea')].forEach(inp => inp.disabled =
true);

 data.topicId = ctx.params.id;

 await createComment(data);

 [...event.target.querySelectorAll('input', 'textarea')].forEach(inp => inp.disabled =
false);
 update(); //презареждаме страницата без redirect, добре би било да кешираме данните
 от текущата страница
 }
}

async function loadTopic(id, onCommentSubmit) {
 const [topic, comments] = await Promise.all([
 _topicData, //кеширането тук, и реално презареждаме само коментарите, а не
 //всичко
 getCommentsByTopicId(id)
]);

 const userData = getUserData();
 const isOwner = userData && userData.id == topic._ownerId;

 return topicCard(topic, isOwner, comments, createSubmitHandler(onCommentSubmit,
 'content'));
}

```

*data.js - load u select - работа със вземане на данни от сървъра*

```
import * as api from './api.js';
```

```

export const login = api.login;
export const register = api.register;
export const logout = api.logout;

const endpoints = {
 topics:
 `/data/topics?select=_id,title,_ownerId&load=${encodeURIComponent('author=_ownerId:users')}`,
 //query - идва от функционалността на сървъра
 topicCount: `/data/topics?count`, //query - идва от функционалността на сървъра
 topicById: (id) =>
 `/data/topics/${id}?load=${encodeURIComponent('author=_ownerId:users')}`, //query - идва от
 функционалността на сървъра
 createTopic: `/data/topics`,
 commentsByTopicId: (topicId) => `/data/topicComments?where=' +
 encodeURIComponent(`topicId="${topicId}"`) + `&sortBy=${encodeURIComponent('_createdOn
 desc')}&load=${encodeURIComponent('author=_ownerId:users')}`, //и сортирай по последно
 добавено
 createComment: `/data/topicComments`
};

export async function getAllTopics() {
 return api.get(endpoints.topics);
}

export async function createTopic(topic) {
 return api.post(endpoints.createTopic, topic);
}

export async function getTopicCount() {
 return api.get(endpoints.topicCount);
}

export async function getTopicById(id) {
 return api.get(endpoints.topicById(id));
}

export async function getCommentsByTopicId(topicId) {
 return api.get(endpoints.commentsByTopicId(topicId));
}

export async function createComment(comment) {
 return api.post(endpoints.createComment, comment);
}

```

## 28. Modular Applications

Common Scenarios and Best Practices

## 28.1. Component Approach

- Components are a common theme among contemporary frameworks (React, Angular, Web Components in JS) and libraries (page.js, express.js)
- Focused on separation of concerns and composability:
  - Combine presentation/style and business logic in a single unit
  - Encapsulate state and control
  - Expose only necessary interfaces
  - Decoupled from the environment (via dependency injection)
  - Highly composable with other components

Всички съвременни framework работят с компоненти.

По област на приложение разделяне на отговорностите.

Капсулираме състоянието и контрола и визуализацията им.

Да можем тези неща да ги използваме на повече от едно място, и да можем да ги композираме.

Да не се налага да променяме на 100 места. Модула с логиката го вкарваме в приложението.

## MVC – Model-View-Controller



10 файла с име завършващо на Model

10 файла със същото име завършващо този път на View

И още 10 файла със същото име завършващо този път на Controller

Това е същото като в една работилница да сложим всичките червени части в една кофа, в друга кофа всичките сини части. Т.е. групирате нещата по признак какво са те, а не какво правят!!!

Когато трябва да променим как се зарежда примерно login страница, ще трябва да бъркнем/променим както в Controller, така и във View и в Model.

За Back-end обикновено работи MVC, защото различни хора се занимават с Model, с View и с Controller. Най-вече view е отделно, models никой не ги пише, а се съсредоточаваме върху Controller.

На Front-End MVC не е приложимо. Или е приложимо, но никак не е практично!!! В back-end една кофа с болтове, гайки и шайби един размер, в друга кофа трите вида болтове, гайки и шайби с друг размер, и т.н.

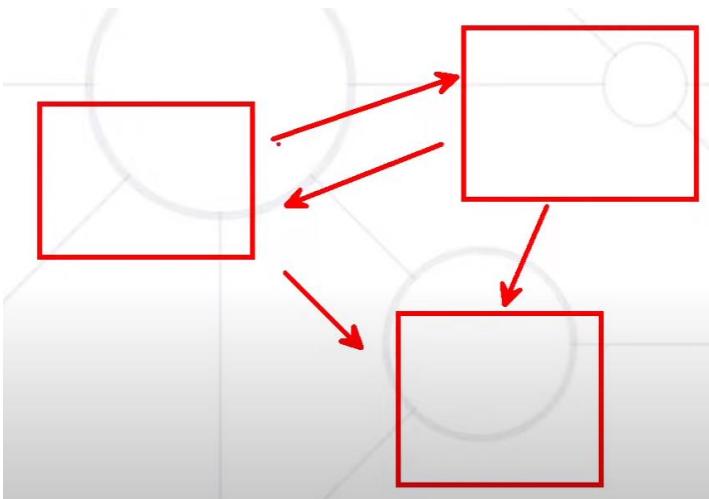
Компонентите от MVC работят на този принцип.

## 28.2. Application State

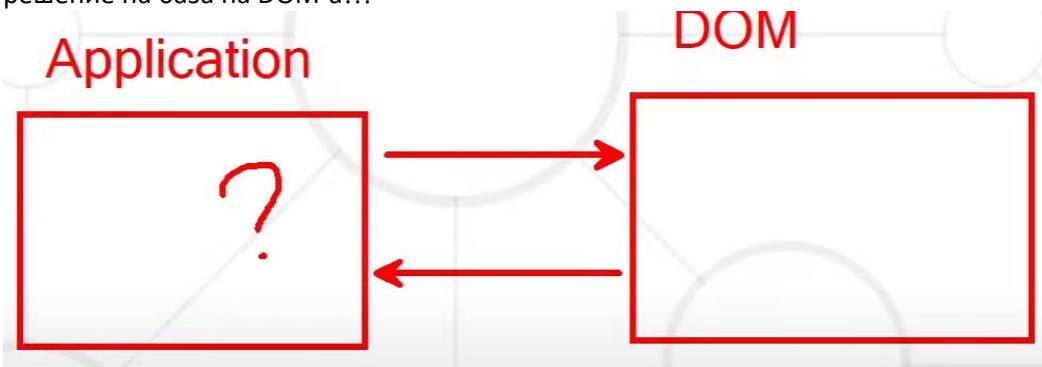
- Avoid storing state in the DOM
- Avoid attempting to infer state from the DOM
  - E.g., using the text content of an HTML element to reconstruct what a database record looked like
- Try to write declarative DOM logic:
  - Describe what the DOM should look like for a given state
  - When the state changes, the DOM follows
  - Rendering libraries allow for efficient DOM redraws – например lit-html

Вместо да изграждаме DOM-а, то описваме за дадено състояние как да изглежда DOM-а. И когато състоянието на приложението се промени, DOM-а го следва.

Кръгова логика – компонентите един спрямо друг си комуникират в двете посоки понякога. При прекалено много връзки и голям проект – става спагети 😊).



Това е лоша практика – ок е DOM да се визуализира на база някаква логика, но не е ок логиката да взема решение на база на DOM-а!!!



### 28.3. Routing

- Attempt to couple application content with the **URL route**
  - This allows more efficient use of **browser history** and **sharing** links to specific parts of the application
  - Can be done with **paths**, **query parameters** or **fragments**
- Examples:
  - **Search terms** should be included as query parameters
  - If a catalog is paginated, include the **current page** in the URL
  - Toggleable content or **sub-navigation** can also be included

### 28.4. Action Feedback – SinglePageApplication gives this

- Provide **instant acknowledgement** for user actions:
  - Change appearance when links and buttons are **clicked**
  - Clear the view on **navigation**
  - Show **loading indicators** during network requests – until директива
  - **Disable input** during requests, to prevent double submission – при формуляр като сме натиснали enter, полетата да станат в сиво недостъпни, докато сървъра не върне резултата от submit-а. За да не събmittнем няколко пъти почти наведнъж.
- Don't overdo feedback:
  - **Don't** attempt to validate input before the user has finished – на всеки символ да му излиза на потребителя, че е срешил е досадно/дразнещо

- There's **no need** to show notifications for everything

## 28.5. User Input

- Always **sanitize** user input / да почистваме кода:
  - Remove leading and trailing **whitespace**
  - Do not automatically include all form data in the request – only pick the **properties** that are **part of the collection**
  - **Prevent** insertion of **HTML** anywhere in your code
  - **Never** use **eval** where user input is involved – но в случай когато server-а като отговор връща не JSON, а JS код, тогава е приложимо. Щом данните идват от сървъра, то е контролирана среда
- Remember that the front-end application **does not** provide security – the **server** must **double check** all user actions – сървърът е човека, който прави финалната проверка и казва дали тези данни могат да влязат.

## 28.6. Error Handling

- Always **anticipate errors** from **network requests** and **user input**
- Errors that can be **resolved automatically** can be handled **behind the scenes**
  - You can **catch** them where they occur
  - E.g., data parsing errors, empty server responses, etc.
- Errors that **concern user action** must be **propagated/размножени** to the **presentation layer** of the app (**rethrow**, or don't catch)
  - E.g., validation errors

## 28.7. Примерен процес/стъпки на писане на проект:

1. Extract project skeleton and examine files
2. Setup NPM project (package.json + libraries)
3. Analyze the HTML and determine templates
4. Configure routing with placeholder modules
5. Implement requests
6. Implement each view
  - 6.1. Create static templates
  - 6.2. Implement fetch requests
  - 6.3. Add parameters to template
  - 6.4. Add event listeners (if any)

## 28.8. Team manager задачата

## 29. Workshop c .fetch() и .then()

Задачата за movies

<https://github.com/movies-softuni/movies-softuni.github.io>

## 30. Exam preparation

node server.js

npm install

npm run start – вместо LiveServer

**npm run test** – рънва тестовете

To make just a single test run, instead of the full suite (useful when debugging a single failing test), find the test and append `.only` after the `it` reference:

```
it.only(`register makes correct API call [5 Points]', async () => {
 const data = mockData.users[0];
 const { post } = await createHandler(endpoints.register, { post: data });
```

On slower machines, some of the tests may require more time to complete. You can instruct the tests to run more slowly by slightly increasing the values for `interval` and `timeout`:

```
5 const host = 'http://localhost:3000'; // Application host (NOT service host - that can be anything)
6 const interval = 300;
7 const timeout = 6000;
8 const DEBUG = false;
9 const slowMo = 500;
```

Note that **interval** values greater than 500 and **timeout** values greater than 10000 are not recommended.

If this doesn't make the test pass, set the value of `DEBUG` to `true` and run the tests again – this will launch a browser instance and allow you to see what is being tested, what the test sees and where it fails (or hangs):

```
5 const host = 'http://localhost:3000'; // Application host (NOT service host - that can be anything)
6 const interval = 300;
7 const timeout = 6000;
8 const DEBUG = true;
9 const slowMo = 500;
```

[href](#) ние си го залагаме/слагаме

## 31. HTML <script> defer and async Attribute

```
<script src="demo_defer.js" defer></script>
<script src="demo_async.js" async></script>
```

The defer attribute is a boolean attribute.

If the defer attribute is set, it specifies that the script is downloaded in parallel to parsing the page, and executed after the page has finished parsing. Ако го има този атрибут то дори да го сложим в head-а, пак браузъра ще зареди този скрипт накрая.

Note: The defer attribute is only for external scripts (should only be used if the src attribute is present).

**Note:** There are several ways an external script can be executed:

- If `async` is present: The script is downloaded **in parallel** to parsing the page, and executed as soon as it is available (before parsing completes)
- If `defer` is present (and not `async`): The script is downloaded in parallel to parsing the page, and executed after the page has finished parsing
- If neither `async` or `defer` is present: The script is downloaded and executed immediately, blocking parsing until the script is completed – в този случай скрипта трябва да е сложен в края на `<body>` то.

## XX. Разни

В JS можем да пишем ООП без класове!!!

Method (procedure, ) -> functions inside class instance или метода е функция, която се използва в контекста на обект!

TypeScript – типизиран език – за писане на JS за големи проекти

JS JavaScript или му казват още **VanilaScript**

В JS държавните библиотеки не са заключени!!! Т.е. можем да добавяме функционалности/свойства/методи

Всичко, което се намира на Prototype е инстанция на класа, а всичко което не е върху прототипа се води статичен метод.

Докато не приключи изпълнението на дадена функция, браузърът не рефрешва информацията.

<https://www.crockford.com/2021.html>

**Array == List == Stack == Queue**

**{ } обект == Associative Array (Map/Dictionary)**

**Map и Set**

<https://www.electronjs.org/>

В JS няма **method** overloading!!!

В JS можем обаче да си създадем **functional overloading**

```
function spliceExplanationNotRealOverloading(start, deleteCount = 0, ...items) {
```

```
 console.log('start = ' + start);
 console.log('deleteCount = ' + deleteCount);
 console.log(items);
```

```
}
```

```
spliceExplanationNotRealOverloading(10);
```

```
start = 10
```

```
deleteCount = 0
```

```
(0) []
```

```
spliceExplanationNotRealOverloading(10, 5);
```

```
start = 10
deleteCount = 5
(0) []
```

```
spliceExplanationNotRealOverloading(10, 5, 60, 70, 80);
start = 10
deleteCount = 5
(3) [60, 70, 80]
```

<https://www.toptal.com/designers/htmlarrows/symbols/> - escapes and special symbols in HTML for visualization

Транспилатор е примерно TypeScript

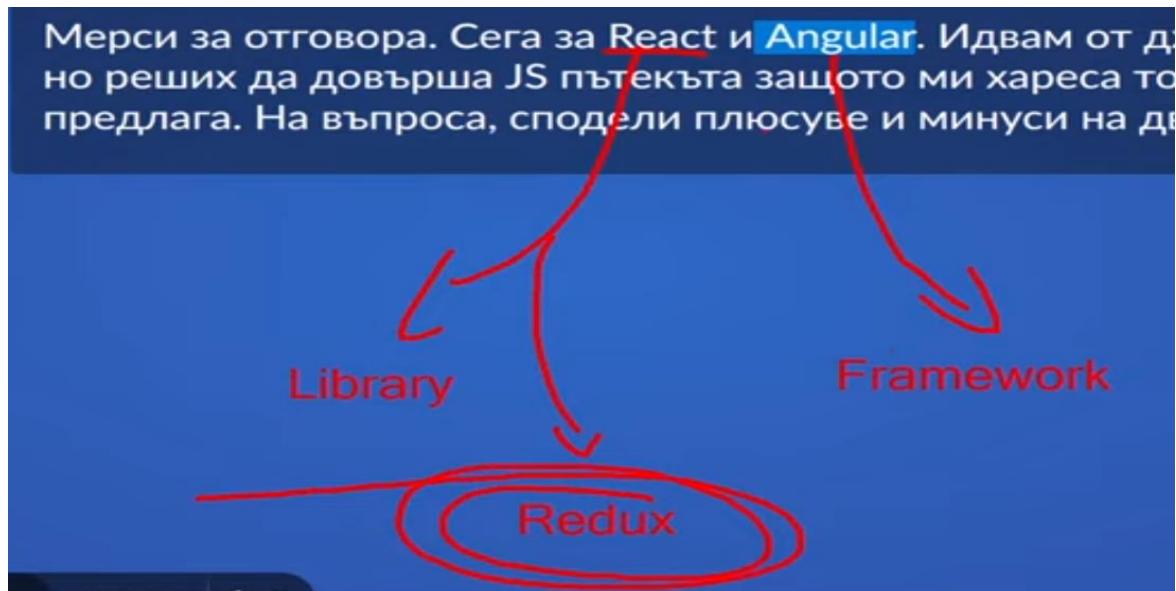
Angular е на TypeScript

```
console.table(values); //печата ги в табличен вид
```

Създай нов масив от 9 елемента, и го запълни с нули

```
new Array(9).fill(0);
```

```
wizardElement.style.backgroundImage = "url('/images/wizard-fire.png')";
```

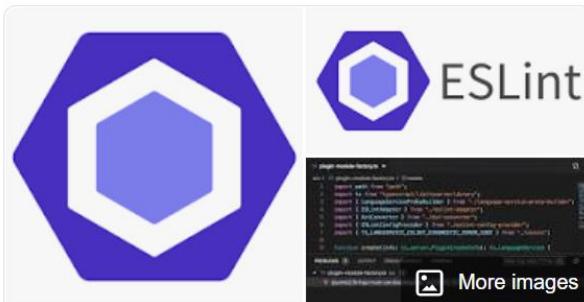


Тези могат да се стилизират  
&hellip; - horizontal ellipsis или просто три точки ...

&lt; стрелка налево

&gt; стрелка на право

## ESLint and Lodash



### ESLint



ESLint is a static code analysis tool for identifying problematic patterns found in JavaScript code. It was created by Nicholas C. Zakas in 2013. Rules in ESLint are configurable, and customized rules can be defined and loaded. ESLint covers both code quality and coding style issues. [Wikipedia](#)



### Lodash



Lodash is a JavaScript library which provides utility functions for common programming tasks using the functional programming paradigm. [Wikipedia](#)