# CSS

For every element set the distance from previous element using margin-top (not padding). In overflow: scroll; margins don’t collapse.

## Positioning

The position can take values:  
- absolute

- static (by default)

- relative (relative to default position)

- fixed

- sticky

The absolute checks the first ascendant (parent) element that have set position property and position the element relatively to this ascendant. If the ascendant with set position not found - removes the absolute element from the entire html page and positions it relatively to page sides using top, right, bottom, left properties.

fixed - same as absolute but always relative to entire html page and scrolls with the scrolling.

sticky - positioned as static but begins to scroll with page when touch the html page edge. If you set e.g. top: 50px – it will begin scrolling when the element be 50px from top edge.

## Inheritance

If you define a css rule like

.class1 .class2 {

this rule will work even if .class2 nested deeper than 1 level and the class2 is not children but e.g. grandchildren of class1.

Some properties are inherited by default no matter how deep nested elements

<div id="box">

    <section>

        <section>This is some text inside the box.<section>

    </section>

</div>

#box {

      background-color: brown;

      padding: 100px;

}

“This is some text…” will also have background color brown. The inheritance works on those properties for all selectors: tags, classes, id.

Some properties like `padding` are not inherited, to inherit them it should be set to `inherit`

#sec {

  padding: inherit;

}

But it inherits from close parent, not from grandparents:

        <div id="box">

            <section>

                <section id="sec" >This is some text inside the box.<section>

            </section>

        </div>

#box {

      background-color: brown;

      padding: 100px;

}

#sec {

  padding: inherit;

}

The padding of #sec will be 0px because the padding of parent <section> is default 0px.

## Select needed element

To select all descendants (children, grandchildren etc.) write

.class \* {

  padding: 10px;

}

To select only first-level children:

.class > \* {

  padding: 10px;

}

Select first elements B directly followed by A elements:

A + B {

  padding: 10px

}

Select all elements B followed by A element (on the same level of nesting):

A ~ B {

  padding: 10px

}

Select first of element and first children(p) of nested elements:

div p:first-child{

  padding: 10px;

}

To select only first child of element use

div > p:first-child{

  padding: 10px;

}

Select every element that is the 8th child of another element:

:nth-child(8){

  padding: 10px;

}

In SCSS to select certain descendant elements instead of

.class div {

padding: 10px;

}

you can write in this way:

.class {

div {

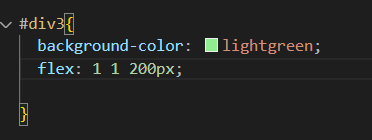
padding: 10px;

}

}

## FlexBox

In flex-box the `flex` property works bad:



It’s ignore 200px size.

So I prefer to use related parameters:

#div1{

  background-color: coral;

  width: 100%;

  flex-shrink: 1;

}

#div2{

  background-color: lightblue;

  width: 200px;

  flex-shrink: 0;

}

#div3{

  background-color: lightgreen;

  width: 200px;

  flex-shrink: 0;

}

The div1 take all the rest place and only div1 is shrink.

Don’t use flex-grow and width.

#div1{

  background-color: coral;

  min-width: 100px;

  width: 50%;

}

#div2{

  background-color: lightblue;

  min-width: 100px;

  width: 100%;

}

#div3{

  background-color: lightgreen;

  width: 200px;

  flex-shrink: 0;

}

div2 is twice wider than div1, but if shrink – 100px both:

## Width Height 100%

If ancestors heights are set to 100% - the elements will take the whole screen space.

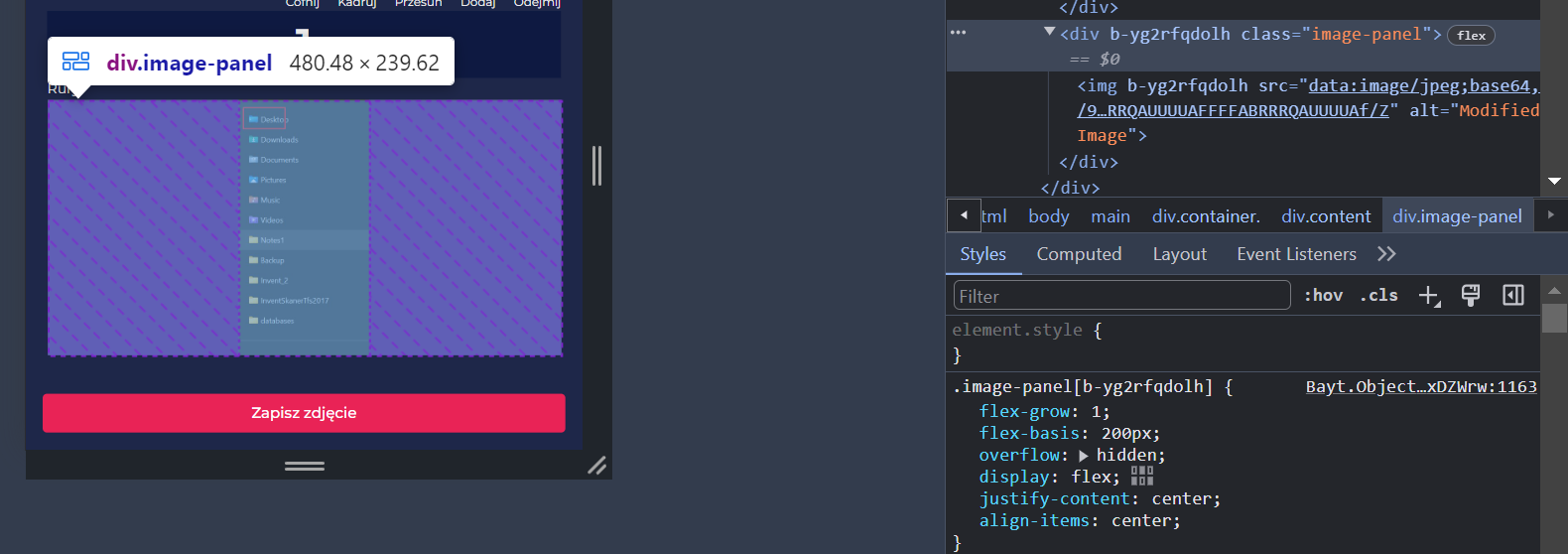
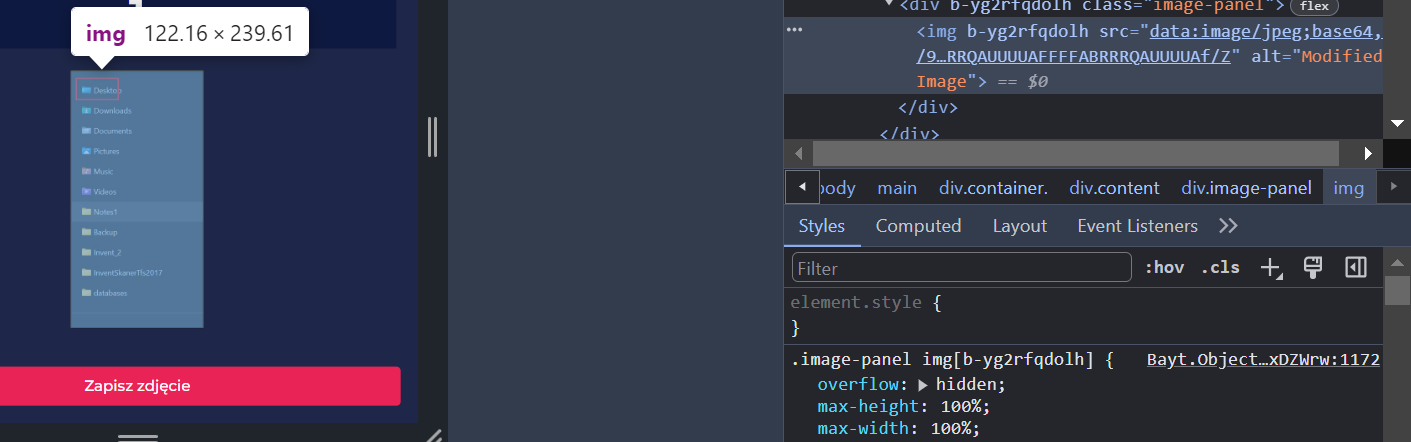
But, if one of nested element are bigger than screen size, and the `overflow` of it parent is not set – it will EXPAND the screen size, and all ancestors will take the 100% of new screen size. This is because the default overflow is `visible`.

Also setting h/w in % works like shit, so try to use these parameters together:

**flex-grow: 1** – sets that the element can grow and how much relatively to other growing elements,

**flex-basis: 100px** – sets the minimal size (height if the parent is flex-direction column)

**overflow: hidden** (or another) ­– limits the size of children.

Then, when you have a growing box, if you put another element inside – you can operate on **max-width max-height** parameters.

## Pseudo classes

::after – adds element (inline) after main element – in the end:

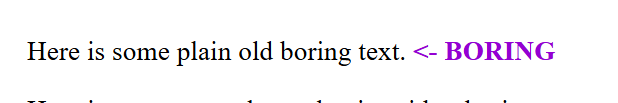
<p class="boring-text">Here is some plain old boring text.</p>

.boring-text::after {

content: " <- BORING";

color: darkviolet;

font-weight: bolder;

}

# SCSS

All variables starts from “$” sign:

$primary-color: #3498db;

$font-size-base: 16px;

.button {

  background-color: $primary-color;

  font-size: $font-size-base;

}

# HTML

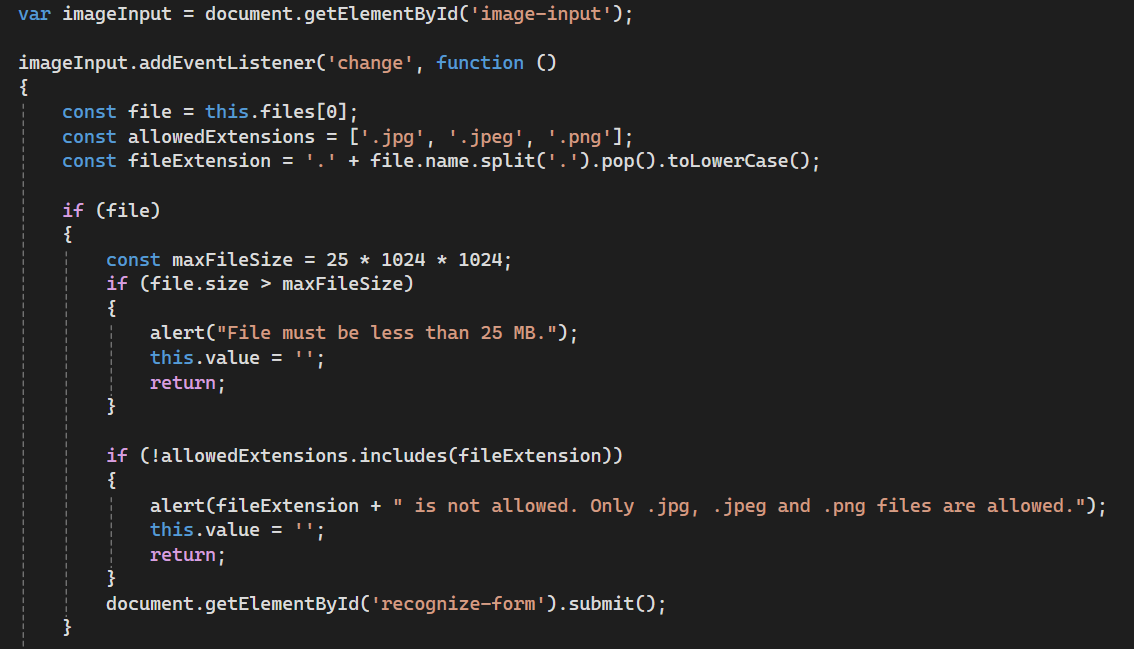
## Form input validation

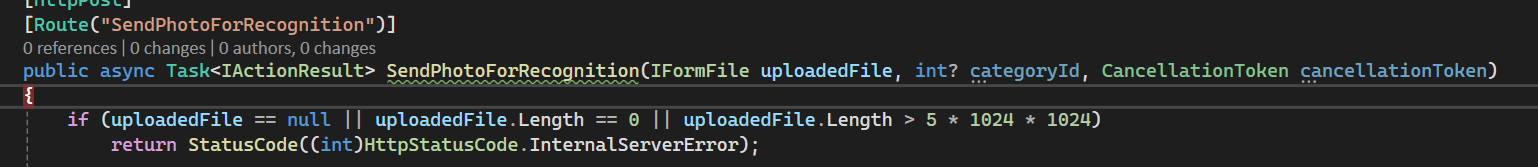
There are a lot of input type like “text”, “password”, “number”, “tel”, “time”

`accept` set file filter when the file explore window is opened, but the user can select any file still.

<label> is bound to `id`.

<input> is bound to controller method parameters through `name`.

Additional fast validation in js:  
  
html and js validation is only on client side and can be avoided. To enhance security add server-side validation:



# JS

What to learn:

* Variables (let, const)
* Functions
* Arrays & Objects
* Loops, if/else
* Scopes
* Basics of this
* Promises & async/await
* ES6+ features (spread operator, destructuring, arrow functions)

## JS Front End

Get element by Id:

const elem = document.getElementById("main-text");

elem.style.color = newColor;

Instead of

<button onclick="onClickFunction()">Click me</button>

It’s recommended always to use addEventLestener() that keeps js separately from html and allows to add multiple onclick functions.

The addEventListener() method sets up a function that will be called whenever the specified event is delivered to the target.

//both versions work fine, but the script must be after pulled html elements – the DOM

// must be generated first (or use document.addEventListener('DOMContentLoaded', () => {)

<script>

document.getElementById("login-form").addEventListener("submit", e => {

    document.getElementById("login-button").disabled = true;

});

</script>

//version 2

<script>

let loginForm = document.getElementById("login-form");

let loginButton = document.getElementById("login-button");

loginForm.addEventListener("submit", e => {

    loginButton.disabled = true;

});

</script>

When DOM is loaded, the browser can safely apply js scripts on html elements, because “DOM loaded” means that all html elements of file are turned out into in-memory nodes that are part of DOM tree. The images, fonts, stylesheets may finish loading **after** DOM.

## Fundamentals of JS

In JS methods names start from small letter.

### Primitive DataTypes:

- Number – any number 3, 2.5, -2. Number bigger than ~1015 loose precision, than use BigInt  
- BigInt numbers (without decimals) that can be bigger than ~1015  
to declare BigInt: let y = 9999999999999999n;  
BigInt can’t be used in arithmetic with Number, use conversion:  
let x = 5n;  
let y = Number(x) / 2;  
- String, can use both “ “ and ‘ ‘.

- Boolean  
- Null  
- Undefined (similar to null)  
- Symbol - A newer feature to the language, symbols are unique identifiers  
Other data types:  
- Object - collections of related data.

- function (it is an object at the same time)

foo = function() {};

console.log(typeof foo); // function

take integer part:

 parseInt(1.9) // 1

parseInt(-3.4) // -3

Difference between var, let, const – it’s not important.

Const only can’t be reassign by “=”, but the value can be mutable.

const person = { name: "Alice" };

person.name = "Bob";           // ✅ OK

person.age = 30;               // ✅ OK — adding new property

// person = { name: "Carol" }; // ❌ Error — reassignment

Var variables have function scope. Let variables has the block scope. It can’t be accessible outside the particular code block ({block}).

**function** f() {

**if** (**true**) {

**let** b = 9

console.log(b); *// 9*

}

console.log(b); *// ReferenceError: b is not defined*

}

f();

console.log(b); *// error*

**function** f() {

**if** (**true**) {

**var** b = 9

console.log(b); *// 9*

}

console.log(b); *// 9*

}

f();

console.log(b); *// ReferenceError: b is not defined*

Best practice is not using var, but just:

**function** f() {

**let** b;

**if** (**true**) {

b = 9

console.log(b); *// 9*

}

console.log(b); *// 9*

}

This is ok:

**let** a = 10;

**function** f() {

a = 9

console.log(a) *// 9*

}

f();

**let** a = 10

*// It is not allowed (but it’s ok for var)*

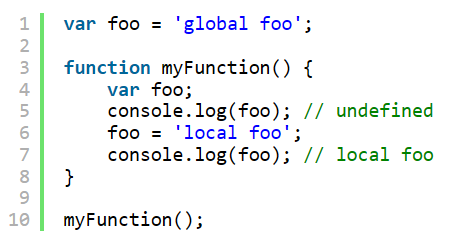
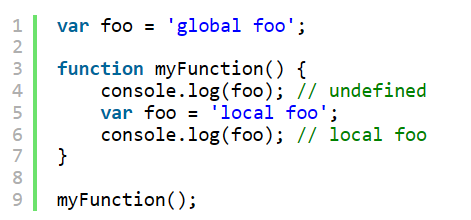
**let** a = 9

*// It is allowed*

a = 10

### Hoisting in JS (only for var):

When JavaScript is executed, the interpreter moves or “hoists” all variable declarations to the top of their containing function / scope boundary, regardless of where they occur.



Transforms into this

This is ok:

a = 10;

**var** a;

console.log(a); *// 10*

a = 10; *// error*

**let** a;

console.log(a);

  
Const is the same as let but can’t be changed.  
  
Instead of

Use (but won’t work with “ “)



Difference between loose equality “==” and strict equality “===”: (also works in TS)  
both treat NaN != NaN,  
“==” performs a type conversion when comparing two things, e.g. these consts are loose equal:  
const num = 0;

const big = 0n;

const str = "0";

const obj = new String("0");

const bool = false;  
“===” doesn’t perform a type conversion, examples above are not strict equal.  
null == undefined, but null !== undefined

null != false

new String('foo') != new String('foo')

Only Object.is(NaN, NaN) return true for comparing NaN (or use custom solutions).

let str1 = new Object("0");

let str2 = new Object("0");

let zero = 0;

console.log(str1 == zero);              // t

console.log(str2 == zero);                                  // t

console.log(str1 == str2);              // f

### Functions

function myFunction(p1, p2) {

    return p1 \* p2;

  }

Functions are hoisted to the top of the scope.

In JS, functions can be:

* Assigned to variables
* Passed as arguments
* Returned from other functions

function sayHi() {

    console.log("Hi");

}

let greeter = sayHi; // Assigning function to variable

greeter(); // Calls sayHi

let a = add(5,5);

function add(a, b) { //functions hoisted to the top of the scope

    return a + b;

}

let a1 = multiply(2, 2); //error

const multiply = function(a, b) { //function expressions don't hoisted

    return a \* b;

};

You can attach properties to functions, because functions are objects:

function counter() {}

counter.value = 0;

console.log(counter.value);

No “this” keyword.

Default parameters as in C#.

//"params" in JS

function sum(...numbers) {

    let sum = 0; //jest let sum; won't work;

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

        sum += numbers[i];

    }

    return sum;

}

console.log(sum(1, 2, 3, 4)); // 10

Functions passed to other functions are common:

//setTimeout is a JS function that runs the passed function after delay (ms)

setTimeout(() => console.log("Delayed"), 1000); //doesn't block the program, just execute this later

Closure – the function that return a function and may have free variables (global variable):

function makeCounter() {

    let count = 0;

    return () => ++count;

}

const counter = makeCounter();

console.log(counter()); // 1

console.log(counter()); // 2

## Keywords

### This

In JavaScript, **this** is dynamic — its meaning depends on who called the function, not where it's defined.

Global scope:

console.log(this); // In browser: Window object

Inside object:

function regularFunction() {

    console.log(this.name);

}

let obj = { name:'Alice', func:regularFunction };

obj.func(); //Alice

Lost content:

const person = {

    myName: "Bob",

    greet() {

        console.log(this.myName);

    }

};

const sayHi = person.greet;

sayHi(); // ❌ undefined

This also prints undefinde:

const person = {

    myName: "Bob",

    greet() {

        console.log(this.myName);

    }

};

let myName = "John";

const sayHi = person.greet;

sayHi(); // ❌ undefined

Because nobody is called sayHi(), must be person.greet, SayHi() won’t work because there is *this* in greet();

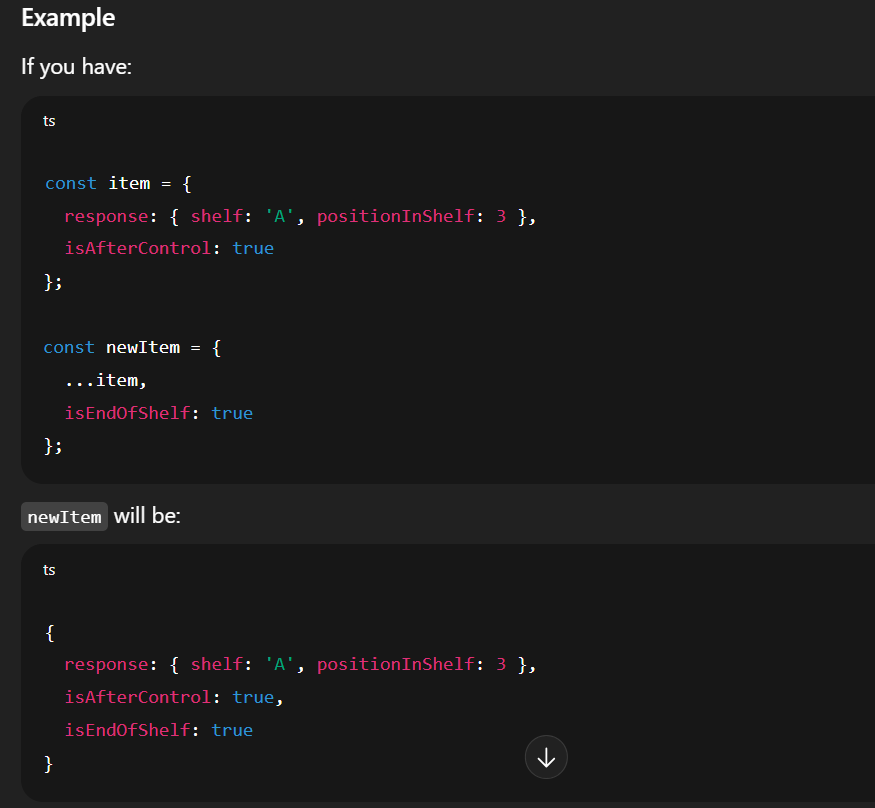
### { …item }

//returns and object same as item but with additional isEndOfShelf property (if exist - replaces the original value)

return {

    ...item,

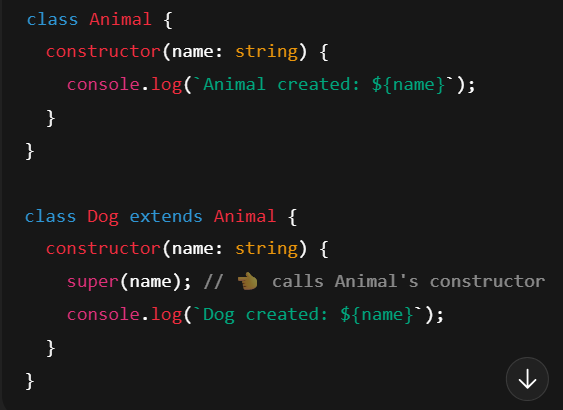
    isEndOfShelf

};

It’s a good practice in Angular to use it, because it creates new object that is easier to detect to update the UI rather than mutation of existing object. Even if the item already has the *isEndOfShelf* property, it’s still better to use this approach and return new object instead of modifying existing one by item.isEndOfShelf = true;

### Super()

The *super()* calls base class constructor:



In a derived class (one using *extends*), if you define a constructor, you must call *super()* before using this.

## Collections

**Arrays**

Arrays in JS:

* + Equivalent to List<T> and array T[] in C#.
  + Dynamic in size
  + Can hold mixed types



**Object**

* + Equivalent to Dictionary<string, object> in C#.
  + Keys are always strings (or symbols), can have methods.

//Object

const person = {

    name: "Alice",

    age: 30

};

**Map**

* + Similar to Dictionary<TKey, TValue> in C#, but
  + Maintains insertion order
  + **Map** is better than **Object** when keys are not strings or when key order matters.
  + Methods .set, .get, .has, .delete, .clear

const map = new Map();

map.set("key", 123);

map.set({ id: 1 }, "value");

console.log(map.get("key")); //123

**Set**

* + Equivalent to HashSet<T> in C#

//Sets

const set = new Set([1, 2, 3, 3]); //.add, .has, .delete

console.log(set); // 1, 2, 3

## Loops

for(let i in array) == for(let i = 0;i < array.length;i++)

c# foreach equivalent: for(let e **of** array)

## Asynchronous programming

### Short tutorial

*Ultrashort tutorial: after more than 3 days of diving in this, there is still no answer how exactly and in which order code executed, there is a mess and I see no articles with real deep dive, they all talk about Event loop, microtask queue and macrotask queue, but in real project there is a mess and I see no exact order. Don’t spend much time on it anymore. If you need, just focus on how to send requests in parallel and how to use “workers” to run code calculations in multiple threads.*

Before you read the article below, you should know that in Angular the button click goes to another macrotask queue with higher priority than macrotask queue with e.g. setTimeout.

Basic knowledge about Event loop <https://javascript.info/event-loop>

setTimeout has a time to delay the code within, this time set when the code within will be added to macrotask queue:  
stack hit the

setTimeout(() => console.log(`setTimeout`), 1000);

    console.log(`A`);

time set to 1000 ms, so the

console.log(`setTimeout`)

will be enqueued in 1 second

DON’T USE rxjs observable and subscribe with await:

WRONG CODE EXAMPLE

  async getValueFromApi(value: string) : Promise<Observable<string>> {

    const url = `${this.localTestApiUrl}?returnValue=${encodeURIComponent(value)}`;

    const resp = await this.http.get(url, { responseType: 'text' }); // api answers in 1 s

    console.log(`getValueFromApi - after await`);

    return resp;

  }

  async asyncFunctionClick(functionId : number){

    setTimeout(() => console.log(`setTimeout`), 0);

    console.log(`A`);

    (await this.getValueFromApi(functionId.toString())).subscribe({

      next: res => console.log(res),

      error: err => console.error(`Error:` + err)

    });

    for(let i = 0;i < 1e9;i++) {} // take less than 1 s

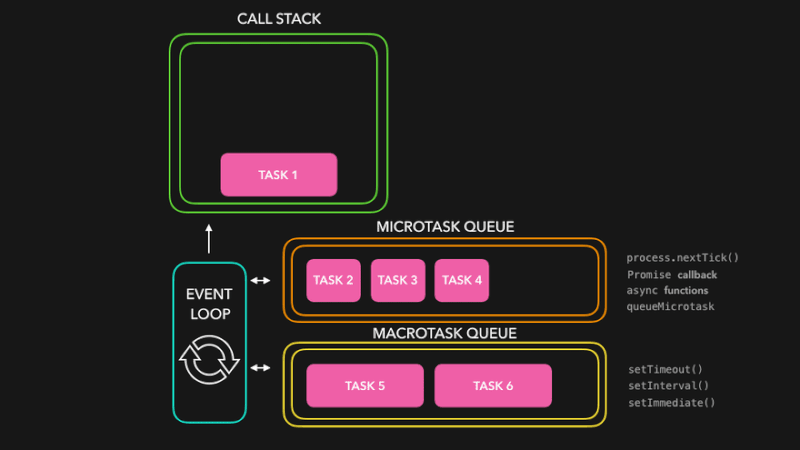
    console.log(`B`);

  }

single click to button

<button (click)="asyncFunctionClick(1)">asyncFunctionClick 1</button>

**Short tutorial**There is only one thread that do all the job by execution Call Stack, the sync code goes straight to Stack, the other tasks are taken from microtask queue and macrotask queue.



After e.g. click on button, the function is executed.  
Sync code executed immediately in Stack, but whet it hit the first await, the executor goes inside await method and after this method finish - all subsequent **sync/awaited** code goes to microtask queue:

   console.log("A");         //executes immediately in Stack

  await printOne(); //executes everything within printOne() and moves subsequent // code to microtask queue

   console.log("B"); //executes after printOne() as part of microtask queue

If new tasks for microtask queue appears – they will be executed before macrotask*.* Example:

Microtask queue*:* [Do1, Do2, Do3]

Macrotask queue*:* [Do\_1, Do\_2]

Execution order: Do1, Do2, Do3, Do\_1, Do\_2

But during execution Do2, another button was clicked that have Do4 and Do\_3 tasks – the new execution order:

Do1, Do2, Do3, Do4, Do\_1, Do\_2, Do\_3

Real example in Angular with button:

<button (click)="printOneClick()">Print one</button>

  async printTwo(){

    for(let i = 0;i < 2e9; i++) {}; // -> Stack

    console.log(2);                 // -> Stack -> 3. 2

  }

  async printOne(){

    console.log(`Entered print one`);       // -> Stack -> 2. Entered print one

    Promise.resolve().then(() => console.log("Promise.then in one")); // -> microtask queue

    await this.printTwo();        // -> Stack goes within printTwo(), after finished await

    //"for(let i = 0;i < 2e9; i++) {};" and "console.log(1);" -> microtask queue

    for(let i = 0;i < 2e9; i++) {};

    console.log(1);

  }

  async printOneClick(){

    console.log("A");                                 // -> Stack -> 1. A

    setTimeout(() => console.log("setTimeout"), 0);   // -> macrotask queue

    for(let i = 0;i < 2e9; i++) {};                   // -> Stack

    Promise.resolve().then(() => console.log("Promise.then")); //-> microtask queue

    await this.printOne();              // -> Stack goes within printOne(), after finished await

                                        // console.log("B"); -> microtask queue

    Promise.resolve().then(() => console.log("Promise.then 2")); // -> microtask queue

    console.log("B"); //already in microtask queue (after await)

  }

  // -> microtask queue execution:

  // 4. Promise.then

  // 5. Promise.then in one

  // 6. 1

  // 7. B

  // 8. Promise.then 2

  // -> macrotask queue execution:

  // 9. setTimeout

}

If you add *await this.printOne();* at the end of *printOneClick()*, it will be scheduled to microtask queue after *console.log("B");* - after first *await this.printOne();* finish.

summary for async/await:

* + awaited code and sync code goes to stack immediately
  + await completion moves subsequent code (in same block) to microtask queue, after any await starts execution of the current microtask queue – there is no sync code to execution.
  + setTimeout always goes to macrotask queue and runs after sync, awaited and microtasks

Due to tutorials every user interaction with UI pushes a new macrotask to macrotask queue. But in real world there might be multiple macrotask queues.

Example queues:

* + Macrotask queue for timers (like setTimeout)
  + Macrotask queue for UI events (like click, hover)
  + Macrotask queue for I/O (input/output – e.g. REST api requests)
  + etc.

In Angular the UI events macrotask queue has higher priority than timers queue.

  async printAClick(){

    console.log("A");

    setTimeout(() => console.log("setTimeout"), 0);

    for(let i = 0;i < 2e9; i++) {};

  }

If you click printAClick and then during execution

for(let i = 0;i < 2e9; i++) {};

click printAClick again, the output will be:

  // A

  // A

  // setTimeout

  // setTimeout

First setTimeout get to queue earlier the second click, but second click goes to priority macrotask queue and when first “for” finishes, it takes the next macrotask form priority queue and executes it.

You can schedule a function execution as a microtask using queueMicrotask(printOne());

Correct code example for async requests to API:

  async getValueFromApi(value: string): Promise<string> {

    const url = `${this.localTestApiUrl}?returnValue=${encodeURIComponent(value)}`;

    const response = await firstValueFrom(this.http.get(url, { responseType: 'text' }));

    console.log(`response received`);

    return response;

  }

  async asyncFunctionClick(functionId : number){

    setTimeout(() => console.log(`af${functionId}: setTimeout`), 0);

    console.log(`af${functionId}: A`);

for(let i = 0;i < 4e9;i++) {}

    const res1 =  await this.getValueFromApi(functionId.toString());

    console.log(res1);

    for(let i = 0;i < 2e9;i++) {}

    console.log(`af${functionId}: B`);

}

The output:

af1: A

af1: setTimeout

response received

Api received and sent back value: 1

af1: B  
  
Because stack:  
1. Schedule setTimeout to macrotask queue

2. Immediately print A

3. executes sync loop

4. goes inside getValueFromApi()

5. send request to API by firstValueFrom() – now the code continuation is delayed. Only when the response come from API – the Network response callback will schedule the microtask with task to continue the code further.

6. Response not come yet, Event loop has nothing to do, it checks microtask queue (empty) than macrotask queue -> prints setTimeout

7. Got response from API -> the subsequent sync/awaited code

    console.log(`response received`);

    return response;

goes as new microtask -> start execute it

8. immediately console.log(`response received`);

9. returned response -> the subsequent sync/awaited code

    console.log(res1);

    for(let i = 0;i < 2e9;i++) {}

    console.log(`af${functionId}: B`);

goes as new microtask -> start execute it

10. printed response

11. loop

12. Printed B

What I click on this button twice? The real output will be   
af1: A

af2: A

af1: setTimeout

af2: setTimeout

response received

Api received and sent back value: 1

af1: B

response received

Api received and sent back value: 2

af2: B

So, why the double setTimeout are not in the end if the first response comes during second click microtask execution? Because “the browser **may schedule a new event-loop tick** immediately for timers before processing the newly created microtasks for the fulfilled Promise (returned response that schedule new microtask to continue)” – chatGPT.   
XD   
So...

It’s look like there is no exact rules and event Loop may violate own rules. It just became too complicated after all JS updates.

### Promises – not important

A **Promise** is an object that represents the eventual **completion (or failure)** of an asynchronous operation and its resulting value. Like Task<T> in C#. Promise is a class, the constructor takes a function as an parameter – this function named **executor**.

A Promise has 3 states:

* **Pending** – still in progres
* **Fulfilled** – completed successfully (then)
* **Rejected** – failed with an error (catch)

const promise = new Promise((resolve, reject) => {//timeout sarts here, not in .then line

    setTimeout(() => {

        resolve("Done!");  // marks as successful or reject("Error!") - marks as failed

    }, 1000);

});

promise

    .then(result => console.log(result)) // if resolved

    .catch(error => console.error(error)) // if rejected

    .finally(() => console.log("Always runs"));

//example

function getUser() {

    return new Promise((resolve, reject) => {

        setTimeout(() => resolve({ name: "Alice" }), 1000);

    });

}

getUser()

    .then(result => console.log(result.name)) // "Alice"

    .catch(err => console.error(err));

The executor runs synchronously when the promise is called, but the *.then* callbacks are **always asynchronous and wait for result, but also wait for current synchronous code finishes.**

**Example:**

console.log("Start");

const promise2 = new Promise((resolve, reject) => {

    for (let i = 0; i < 1e9; i++) {}  // ~1 second

    console.log("Inside executor");

    resolve("Done!");

})

.then(r => console.log(r)); //then schedules microtask

// microtasks are run always after the current synchronous code

for (let i = 0; i < 1e9; i++) {}  // ~1 second

console.log("End");

//Output

// Start

// Inside executor   <-- after 1 s

// End               <-- after 1 s

// Done!

Promise.all([promise1, promise2]);

try{

    await Promise.all([

        resolvePromise.then((result) => console.log(result)).catch((error) => {throw new Error(error);}),

        rejectPromise.then((result) => console.log(result)).catch((error) => {throw new Error(error);})

    ]);

} catch (e){

    console.log(`Caught error with message: ${e.message}`);

} finally {

    console.log("End of try-catch");

}

//Output if timeout resolvePromise is longer than rejectPromise:

// Caught error with message: bad request

// End of try-catch

// ok

**Current stack, microtask queue, macrotask queue**

*.then* or *.catch* are scheduled to microtask queue that runs after current stack (synchronous code finishes). All *setTimeout()* are scheduled to macrotask queue that runs after the whole microtask queue run.

console.log("Start");

setTimeout(() => Promise.resolve().then(() => console.log("setTimeout Promise.then")), 0);

setTimeout(() => console.log("setTimeout"), 0); // macrotask queue

Promise.resolve().then(() => console.log("Promise.then")); // microtask queue

console.log("End");

//Output

// Start

// End

// Promise.then

// setTimeout(Promise.then)

// setTimeout

### Async await

Async/await is syntax sugar over Promises. Async await allows to replace this:

promise.then(result => {

    // ...

  }).catch(err => {

    // ...

  });

To this:

try {

    const result = await promise;

// ...

  } catch (err) {

    // ...

  }

*async* function always return a promise (even if you return a simple value – implicit conversion).

*await* pauses execution inside an async function until a Promise settles:

* + If resolved → returns the result.
  + If rejected → throws the error.

*await* only works inside async functions.

async function getData() {

    return 42;

  }

  getData().then(console.log); // logs 42

  //Behind the scenes: return 42; is turned into return Promise.resolve(42);

To handle errors use try-catch instead of .catch()

Don’t mix await and .then,:

//instead of this

const data = await fetch(url).then(res => res.json());//note that await is applied for whole .then chain - awaits for fetch and then awaits for res.json()

//write this

const res = await fetch(url);

const data = await res.json();

//Full flow example TS

async function getUser(id: number) {

    const res = await fetch(`/api/users/${id}`);

    if (!res.ok) throw new Error("User not found");

    return res.json();

}

async function main() {

    try {

        const user = await getUser(1);

        console.log("User:", user);

    } catch (e) {

        console.error("Error:", e);

    }

}

In modern environments (ES modules, Node 14+, browsers), **top-level await** is allowed

const res = await fetch("...");

But in older environments or files, it must be **inside an async function**.

### Stack, microtask queue, macrotask queue – not important, too complicated

There is Event Loop Flow:  
[Stack] -> [microtask queue] -> [1 from macrotask queue] ->

[Stack] -> [microtask queue] -> [ 1 from macrotask queue] ->

…

The app runs it indefinitely.

But when the sync code of Stack heats to await – it schedules the subsequent code to microtask queue (that may already have entrants – the subsequent code will be after them) and runs everything within await as it was at the beginning of microtask queue.

console.log("A");                                           //stack

Promise.resolve().then(() => console.log("Promise.then"));  //schedules to microtask queue

console.log(await getOne()); //run getOne() and await result - moves subsequent code to microtask queue

console.log("B");

// A

// 1

// Promise.then

// B

So, the real loop usually looks like

[Stack] -> [microtask queue] -> [1 from macrotask queue] -> [microtask queue] -> [1 from macrotask queue] ->

Real world example in Angular web page. Each user when enter angular web page has his own angular entity in browser (separated from another users).

When user clicks button

“The angular (browser) schedules the macrotask to the macrotask queue”, but actually it works like independent onClick execution and the next click will start execute only after the first click finish it code execution.   
The first click starts his microtask (onClick body).*The click sends macrotask to queue, but when it start execution it starts his microtask.* It schedules microtask queue, every await schedules subsequent code to same microtask queue and every function/awaited function may schedule their macrotasks to macrotask queue, that will be executed after microtask queue is finished.

So, it have own loop:

[microtask queue] -> [1 from macrotask queue] -> [microtask queue] -> [1 from macrotask queue]

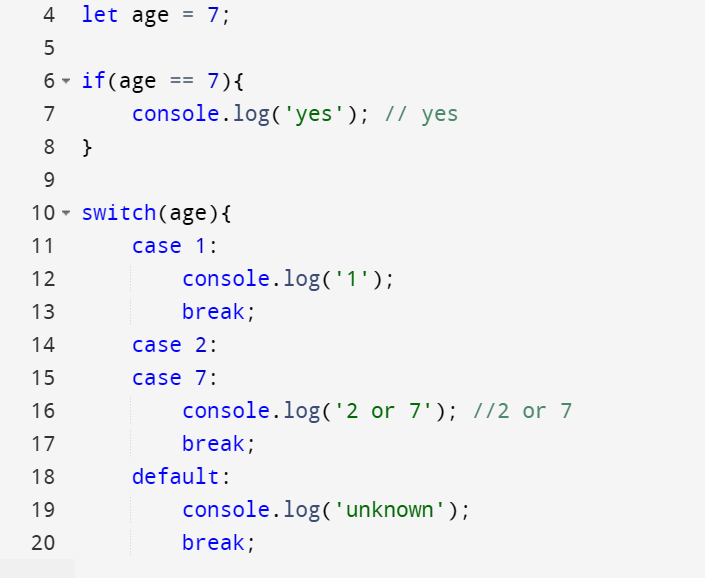
Which will result in

[microtask queue] -> [macrotask queue] -> end.

## C# and JavaScript/TypeScript

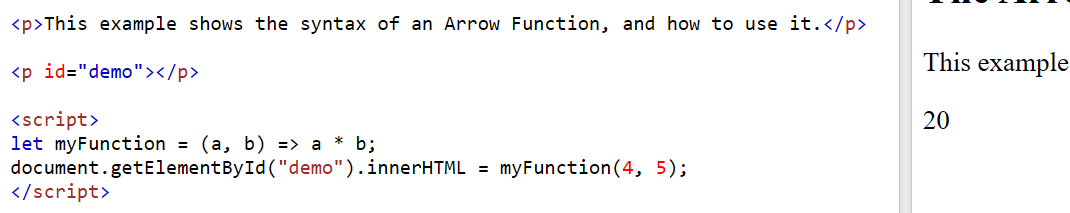
Nice article about C# and JS <https://mauricebutler.wordpress.com/2011/11/07/getting-started-with-javascript-with-a-c-background/> (but it’s from 2011)

- Same { } ;  
- Same if, else, switch



- Class and Interface keywords same in C# and TS

- same accessors (public, protected …) in C# and TS

- arrow => same only for lightweight functions

- async await. C# returns Task, JS and TS return Promise,

- garbage collection,

- C# have NuGet package manager, for JS similar role plays npm,

- TS have union types,

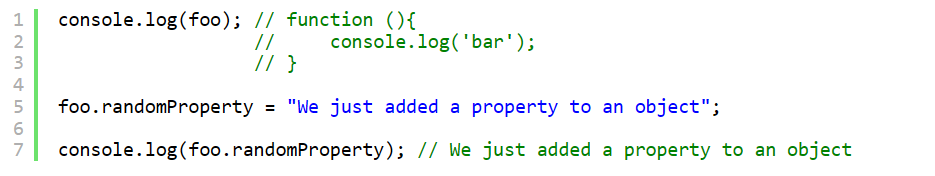
- TS have decorators, e.g. @sealed. Some of them are implemented in C# as keywords or attributes,

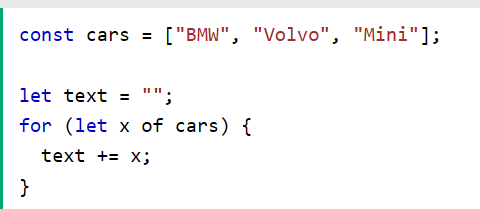
- In JS, the value `NaN` (Not a Number) occurs when the value is returned as a number type, but the value is not parseable as a number. The value `infinity` occurs when a number exceeds the upper limit 1.7976931348623157E+10308,

- The type of `Null` in JS is `object`,

- In JS anything that exists and has a value will evaluate as true unless the value is false, null, undefined, 0, NaN or an empty string,

- Instead of `someObject?.prop` in JS write ` if (someObject) ` But in TS you can write `someObject?.prop` -> if someObject is null it will return *undefined*.

- You can add properties to objects (in this case it’s a function object):

- C# foreach loop in JS: **in -> of**

- JS **in** iterates over properties:



x: fname, lname, age

Similar try-catch, *error* instead of *exception*:

try {

    throw new Error("Error message");

  } catch (err) {

    console.error("Error caught:", err.message);

  } finally {

    console.log("Try-catch run");

  }

Same as in C#, try-catch works only with sync code or async with await:

const resolvePromise1 = new Promise((resolve, reject) => {

    setTimeout(() => resolve("ok"), 1000 );

});

const rejectPromise1 = new Promise((resolve, reject) => {

    setTimeout(() => reject("bad request"), 1000);

});

try{

    await resolvePromise1.then((result) => console.log(result))

        .catch((error) => { throw new Error(error); });

    await rejectPromise1.then((result) => console.log(result))

        .catch((error) => {throw new Error(error)});

} catch (e) {

    console.log(`Caught error with message: ${e.message}`);

} finally {

    console.log("End of try catch");

}

//the output appears in 1 second:

// ok

// Caught error with message: bad request

// End of try-catch

Instead of “… ?? False” use double negation “!! ...”

!!this.listItemData().group?.some((item) => item.isEndOfShelf)

If group is null or undefined, the !! converts it undefined -> true -> false

!value converts to true only:

console.log(!false);       // true

console.log(!0);           // true

console.log(!"");          // true

console.log(!undefined);   // true

console.log(!null);        // true

console.log(!NaN);         // true

other values are converted to false. Same but opposite for “if” statements like “if(undefined)

## Display pdf on desktop and mobile

1. Download and configure pdfjs library.
2. Use code:

**Code**

@model Invent.Repository.Model.View.InfoKierowcaViewModel

@using System;

@{

    Layout = null;

    var base64 = @Model.TemplatedPDFBase64;

}

<style>

    .iframe-container {

        overflow: hidden;

        position: relative;

    }

        .iframe-container iframe {

            border: 0;

            height: 100%;

            left: 0;

            position: absolute;

            top: 0;

            width: 100%;

        }

        .iframe-container object {

            border: 0;

            height: 100%;

            left: 0;

            position: absolute;

            top: 0;

            width: 100%;

        }

    .btn-space {

        margin-right: 3px;

    }

</style>

<input type="hidden" id="pdfFile" value="@base64" />

<div class="row">

    <div class="row text-center">

        <div class="btn-group" role="group" style="margin-bottom:4px;">

            <button class="btn btn-sm btn-space" id="prev">Poprzednia strona</button>

            <button class="btn btn-sm btn-space" id="next">Następna strona</button>

        </div>

        <div class="col-sm-12">

            <span>Strona: <span id="page\_num"></span> / <span id="page\_count"></span></span>

        </div>

    </div>

</div>

<div class="iframe-container text-center">

    <canvas style="width:100%; max-width:800px" id="the-canvas"></canvas>

</div>

<script>

    (function ()

    {

        let BASE64\_MARKER = ';base64,';

        let pdfjsLib = window['pdfjs-dist/build/pdf'];

        pdfjsLib.GlobalWorkerOptions.workerSrc = window['pdfjs-dist/build/pdf.worker/src/pdf.worker.js'];

        document.getElementById('prev').addEventListener('click', onPrevPage);

        document.getElementById('next').addEventListener('click', onNextPage);

        let pdfAsDataUri = $('#pdfFile').val();

        let pdfAsArray = convertDataURIToBinary(pdfAsDataUri);

        let pdfDoc = null;

        let pageNum = 1;

        let pageRendering = false;

        let pageNumPending = null;

        let scale = 3;

        let pdfCanvas = document.getElementById('the-canvas');

        let ctx = pdfCanvas.getContext('2d');

        pdfjsLib.getDocument(pdfAsArray).promise.then(function (pdfDoc\_)

        {

            pdfDoc = pdfDoc\_;

            document.getElementById('page\_count').textContent = pdfDoc.numPages;

            renderPage(pageNum);

        });

        function convertDataURIToBinary(dataURI)

        {

            let base64Index = dataURI.indexOf(BASE64\_MARKER) + BASE64\_MARKER.length;

            let base64 = dataURI.substring(base64Index);

            let raw = window.atob(base64);

            let rawLength = raw.length;

            let array = new Uint8Array(new ArrayBuffer(rawLength));

            for (let i = 0; i < rawLength; i++)

            {

                array[i] = raw.charCodeAt(i);

            }

            return array;

        };

        function renderPage(num)

        {

            pageRendering = true;

            // Using promise to fetch the page

            pdfDoc.getPage(num).then(function (page)

            {

                let viewport = page.getViewport({ scale: scale });

                pdfCanvas.height = viewport.height;

                pdfCanvas.width = viewport.width;

                // Render PDF page into pdfCanvas context

                let renderContext = {

                    canvasContext: ctx,

                    viewport: viewport

                };

                let renderTask = page.render(renderContext);

                // Wait for rendering to finish

                renderTask.promise.then(function ()

                {

                    pageRendering = false;

                    if (pageNumPending !== null)

                    {

                        // New page rendering is pending

                        renderPage(pageNumPending);

                        pageNumPending = null;

                    }

                });

            });

            // Update page counters

            document.getElementById('page\_num').textContent = num;

        }

        function queueRenderPage(num)

        {

            if (pageRendering)

            {

                pageNumPending = num;

            } else

            {

                renderPage(num);

            }

        }

        function onPrevPage()

        {

            if (pageNum <= 1)

            {

                return;

            }

            pageNum--;

            queueRenderPage(pageNum);

        }

        function onNextPage()

        {

            if (pageNum >= pdfDoc.numPages)

            {

                return;

            }

            pageNum++;

            queueRenderPage(pageNum);

        }

    })();

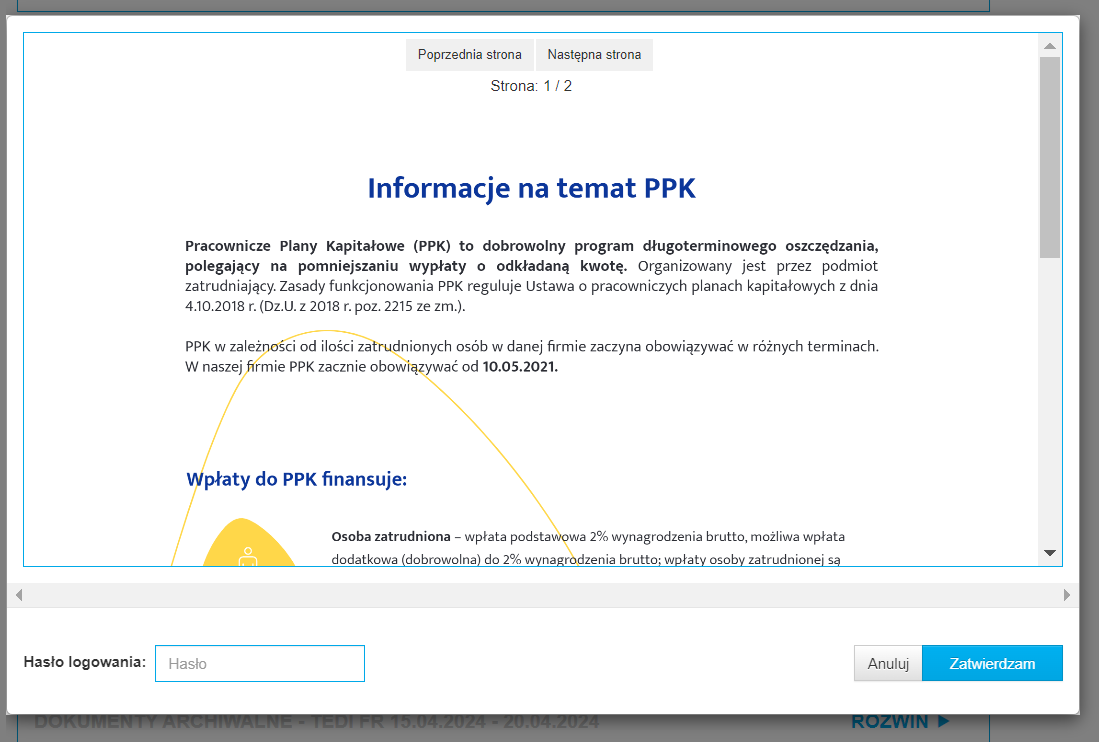
</script>

Where TemplatedFile is

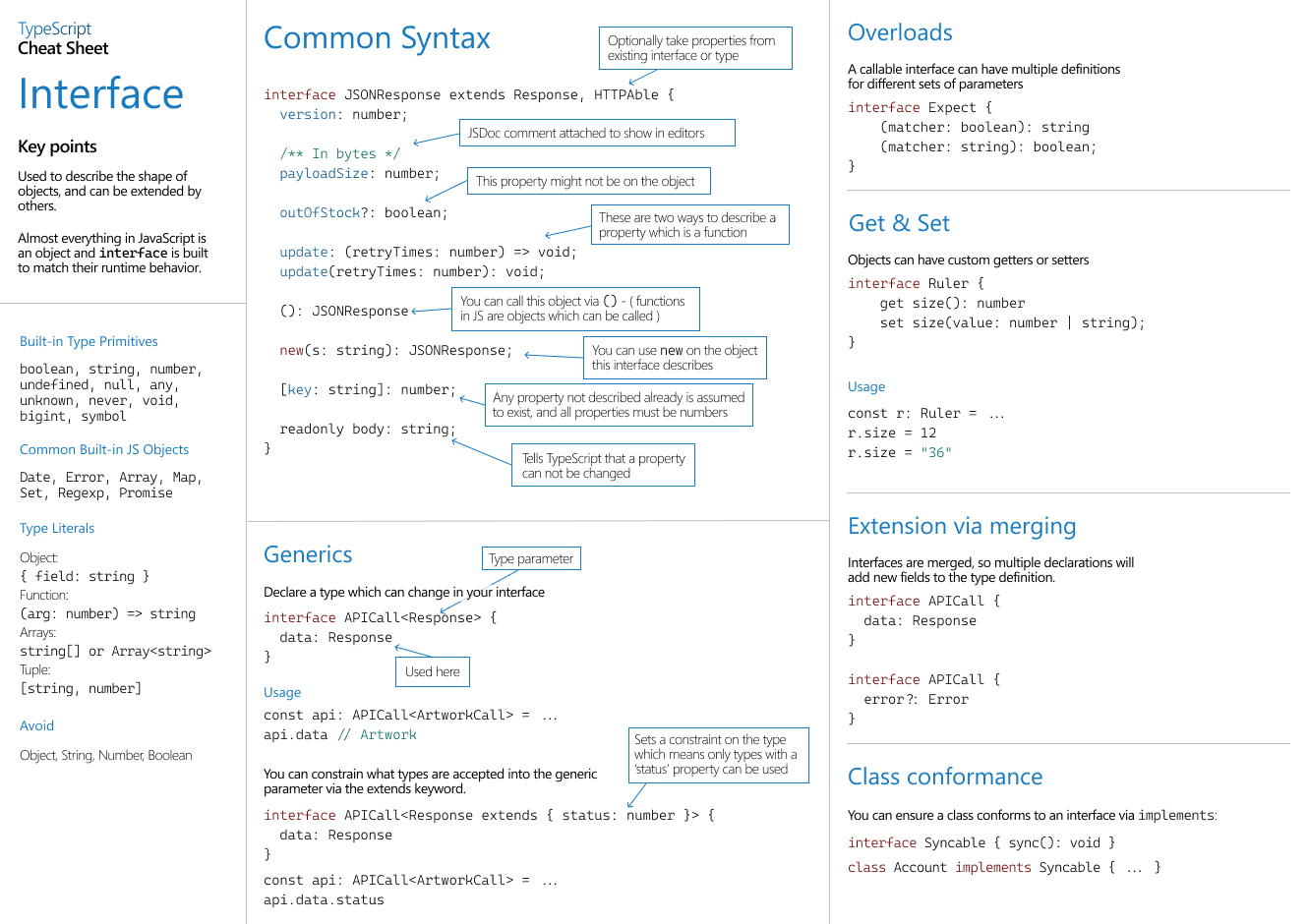
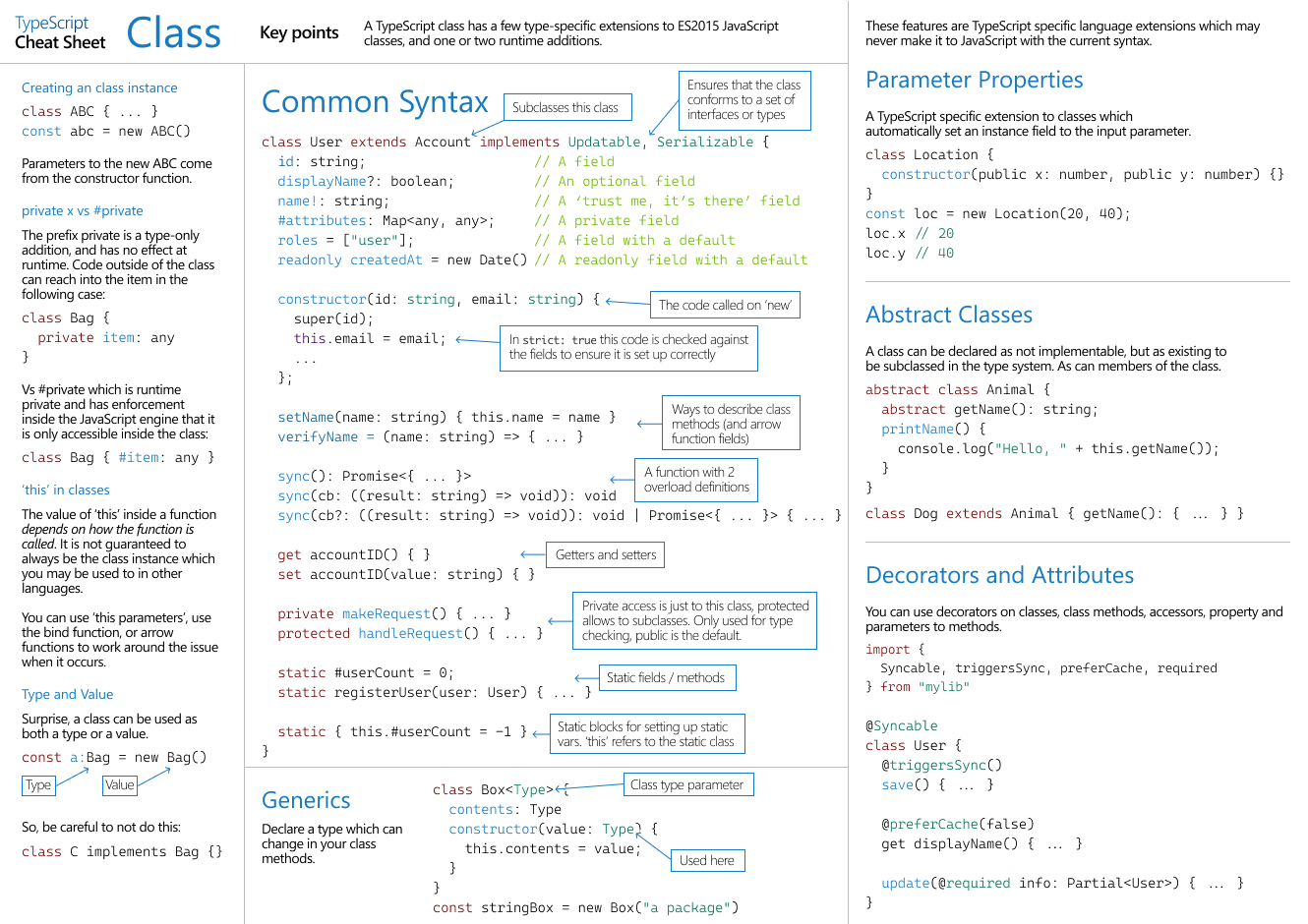
 $@"data:application/pdf;base64," + Convert.ToBase64String(bytes);

bytes is byte[]

Result:



# TypeScript



## TS and JS

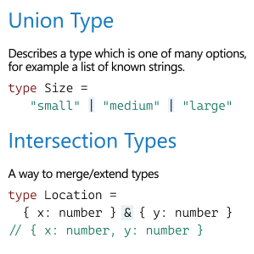
TypeScript is a **superset of JavaScript, most JS code works in TS, but e.g. this code won’t work in TS:**

let a = { Name: "Daniel" };

let b = null;

a.Father = b; // Property 'Father' does not exist on type…

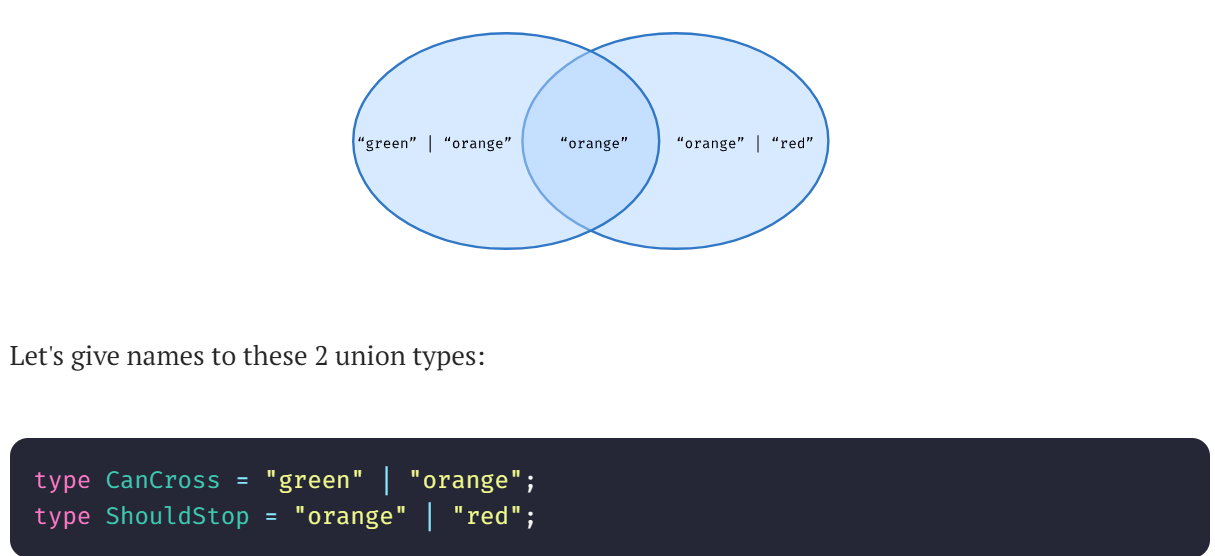
## Types



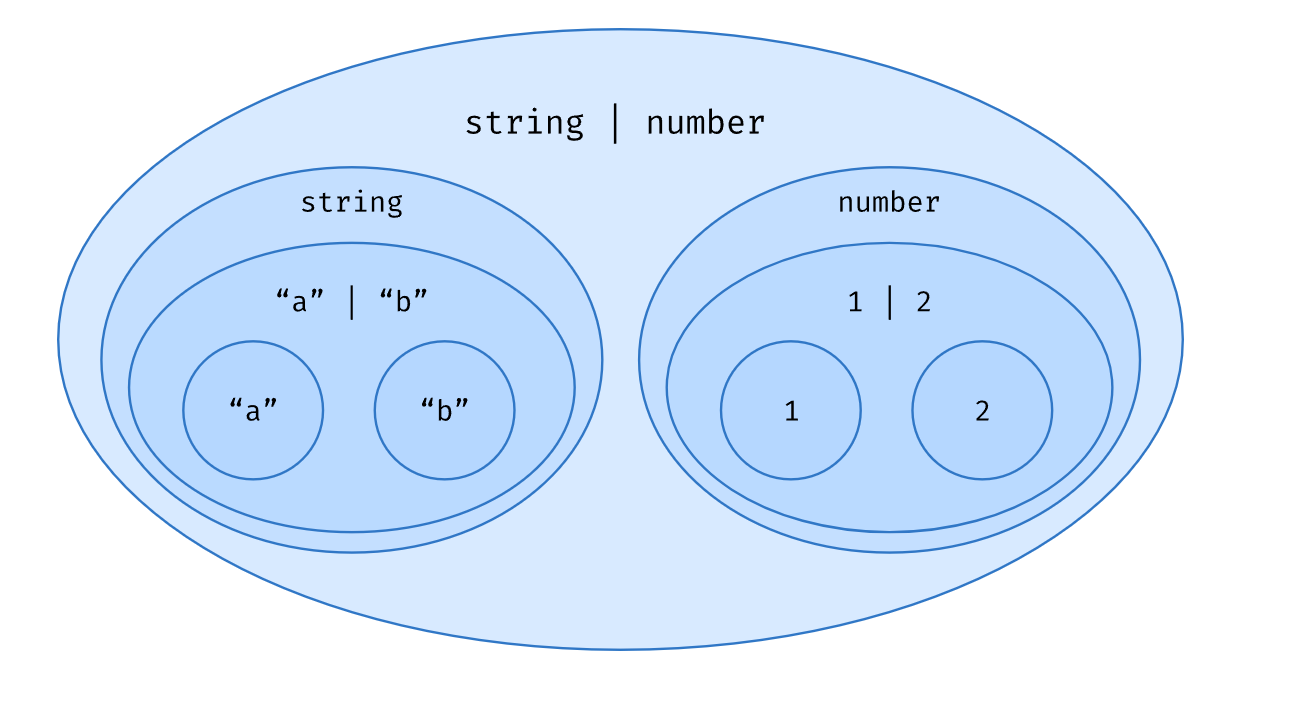
Types: Type aliases can be used to define unions, intersections, and more complex types, but they cannot be extended or implemented. They are generally more flexible for creating custom types.

Nice article about C# and TypeScript: <https://www.typescriptlang.org/docs/handbook/typescript-in-5-minutes-oop.html>

Types are just sets:



In TypeScript, this becomes very natural once you realize that every type is just a set. How do you describe a value that either belongs in the string set or the number set? It simply belongs to the union of those sets: string | number.

- Object doesn’t need to have a relation to interface if I want to use this object in place where suppose to be interface-implemented-object.

When you write:

setTimeout(() => {

if (this.el.nativeElement && this.el.nativeElement.setFocus) {

this.el.nativeElement.setFocus();

}

}, 0);

The setTimeout(..., 0) delays the execution until after the current call stack is cleared. This is often used to ensure that any view updates or DOM manipulations have been completed before executing the focus method. This technique ensures that any pending UI or event processing is completed before the callback runs.

The condition if (this.el.nativeElement && this.el.nativeElement.setFocus) checks that the element exists and **that it has a setFocus method (it’s not calling method).**

Method:

getGlobalStyle(type: 'none' | 'red' | 'black' | 'hero' | 'no-network'): string {

returns string

requires an argument “type” which must be one of: 'none' | 'red' | 'black' | 'hero' | 'no-network'

## Interfaces

export interface SimpleArticleDto {

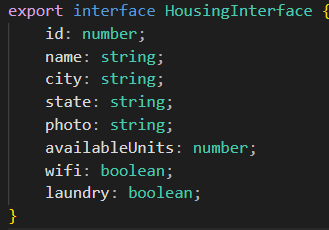
  id: number;

  ean: string;

  priceEntryType?: PriceEntryType;

}

*priceEntryType* is optional.

In TypeScript, both types and interfaces serve similar purposes and can often be used interchangeably for defining the shape of objects

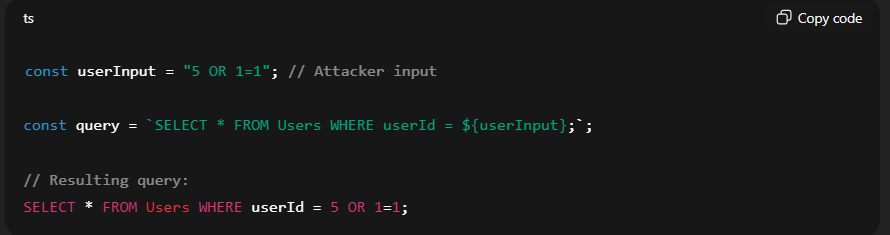
Interfaces: Interfaces can be extended by other interfaces using the “extends” keyword and implemented by classes using the “implements” keyword. This allows you to build on existing interface definitions.

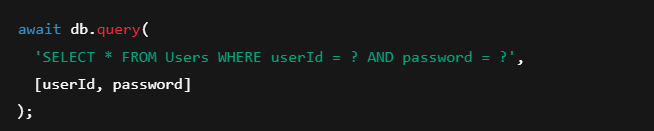
Interfaces: Interfaces are generally used to describe the shape of objects, and their properties cannot be marked as **readonly**.

Types: Type aliases can be used with the **readonly** modifier to create **readonly** types, which enforce immutability for object properties.

# SQL in TS/JS

## Safe using of raw sql queries, sql injections

Instead of (this is an example of sql injection that may appear in when you use $(variable) )

Use

It sends the SQL structure (SELECT \* FROM Users WHERE userId = ? AND password = ?) to the DB and sends the values separately as parameters. The DB engine binds the values to the placeholders safely.

# Angular

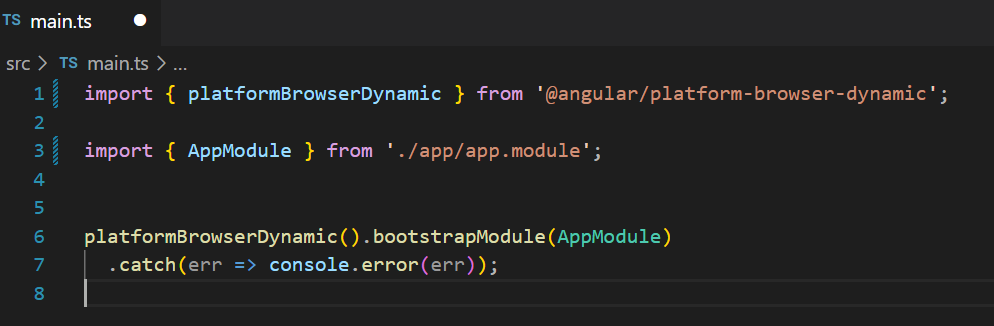
Naming convention – add relative type at the end of entity:

## NgModules, Components

NgModules

Other NgModules Services for DI

Component Component

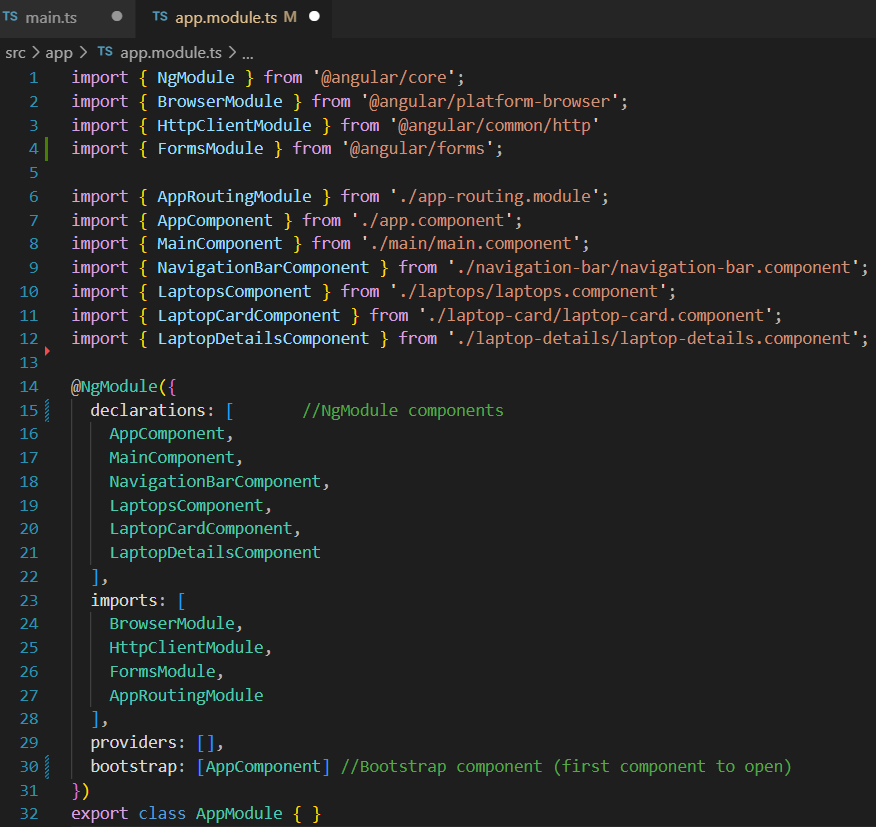
Every application has a **Root Module** that enables bootstrapping, and typically has many more feature modules. A NgModule can associate its components with related code, such as services, to form functional units. NgModules are loaded by lazy-loading.

Main file:

*Import { AppModule } from ‘./app/app.module’* – the ./ means same folder as file with *import.*

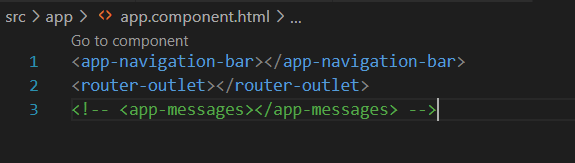
*AppModule* is a class that exported in *app.module.ts*

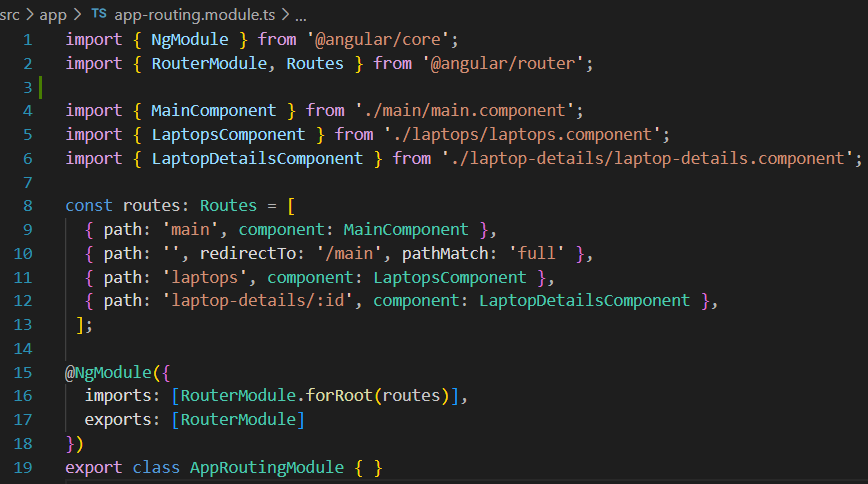
Root Module:



In Components you can add other components in `imports: ` (not in `declarations: `) if imported component is standalone (standalone: true, )

In provider you can enter services for DI (remember about imports)

Root module has rout ngModule named AppRoutingModule and components. App Component has

app-navigation-bar is a component, router-outlet is a directive from RouterModule, that is imported in AppRoutingModule:

The decorators like @Component() determine the type of file. Decorators allow to attach specific kinds of metadata to classes, so that the system knows what those classes mean and how they should work.

NgModules help to give access to components selectors among these components:



HeroesModule – name of this module

- imports: [] is for bringing in other modules (like *CommonModule*, *FormsModule*, or another feature module).

If you want to use a component declared in another module, you must export it from that module and import that module into yours.

@NgModule({

  declarations: [HeroesComponent],

imports: [CommonModule],

  exports: [HeroesComponent, CommonModule]   // <--- make it public

})

export class HeroesModule {}

@NgModule({

  declarations: [DashboardComponent],

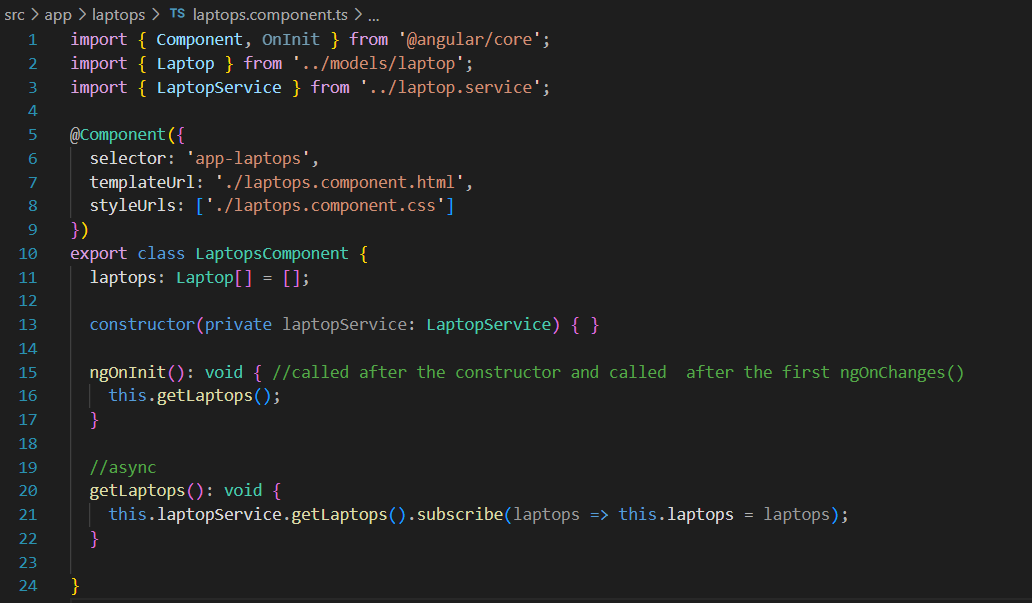
  imports: [HeroesModule]      // <--- brings in HeroesComponent

})

export class DashboardModule {}

Now *DashboardComponent* can use <app-heroes></app-heroes> in its template. *CommonModule* is also available in DashboardModule.

* + Components, directives, and pipes that *HeroesModule* declares are not automatically visible outside.
  + They become visible only if HeroesModule also puts them in its *exports* array.

Laptop component:

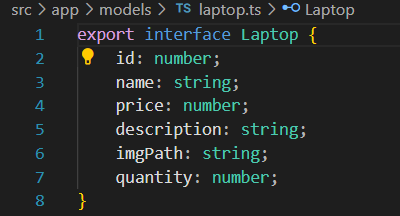
export class LaptopsComponent {

allows to do import and use class methods:

import { LaptopsComponent } from './laptops/laptops.component';

Injection the laptopService:

constructor(private laptopService: LaptopService) { }

Laptop:

laptopService: Services has decorator @Injectable for Dependency Injection



laptops.component.html represents laptops (declared in the .ts file) using directive \*ngFor and represents each laptop as app-laptop-card component:

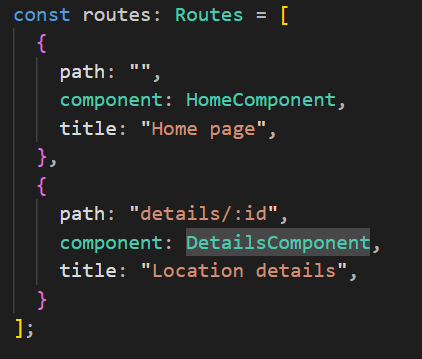
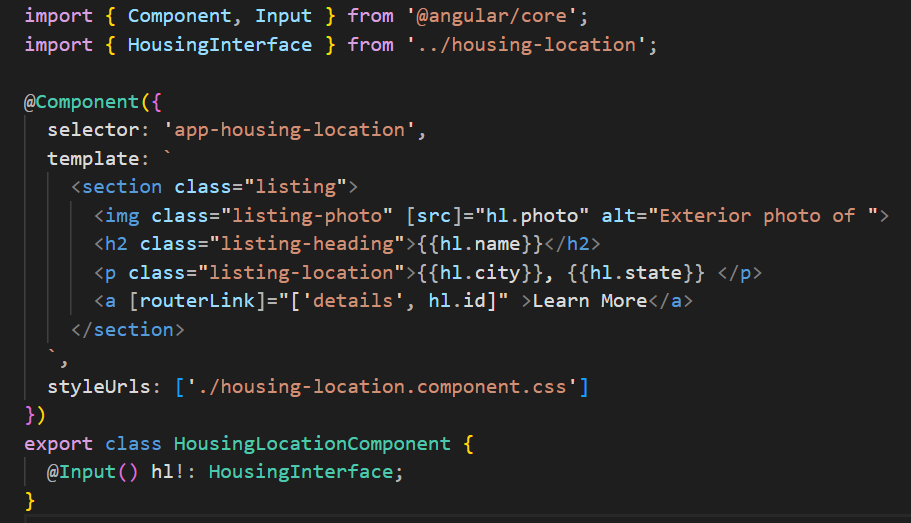


Figure 1 Routes

Injecting components into component:

Main component:

Subcomponents:

### @Directive decorator

Component is a Directive with a template. The @Directive is a decorator that lets you create a custom behavior that can be attached to DOM elements (html elements) or components.

Directive main types:

1. Component
2. Attribute Directive
3. Structural Directive

Examples:

1. Simple attribute directive
2. Also Attribute directive, but conditional:  
   3. Custom structural directive



7.1 In @Directive custom logic for this custom \*ngIf

## Angular commands. Run app.

How to create and run angular app:

Check angular and node version on computer: > ng version

> ng new App-name

> cd App-name

> ng serve --open

The ng serve command:

* Builds the application
* Starts the development server
* Watches the source files
* Rebuilds the application as you make changes

The --open flag opens a browser to http://localhost:4200

Create a component:

> ng generate component *name* { --skip-tests || --inline-template }

Service/interface:

>ng generate service *name*

>ng generate interface *name*

> ng generate module *name* {--flat to not create directory}

## DI

There are five DI types:

@Injectable() // Default, must be provided in a module

@Injectable({ providedIn: 'root' }) // Singleton, shared across the entire app

@Injectable({ providedIn: 'platform' }) // Shared across all Angular apps

@Injectable({ providedIn: 'any' })      // New instance per lazy module

@Injectable({ providedIn: SomeModule }) // Only available if the module is loaded

The first one requires the service to be registered in module “Providers”, also if you import module with service in providers to another module – the second module will also be able to use it. Example:

@Injectable()

export class MyService {

  getMessage() {

    return 'Hello from MyService';

  }

}

@NgModule({

  providers: [MyService], // no need for “exports:” – it’s for components

})

export class SharedModule { }

@NgModule({

  imports: [SharedModule]

})

export class FeatureModule { } //can use MyService

@Injectable({ providedIn: 'root' })

This is a singleton service that don’t need to be registered, just inject it to constructors and use within whole project. The best practice: use this type for every service (@chatGPT)

@Injectable({ providedIn: 'any' })      // New instance per lazy module

Provide a **new instance** of this service **in each module** that injects it — unless that module is eagerly loaded and the service is already instantiated.

## Signal Computed Effect

**Signal**

signal - reactive getter function. It’s like a reactive property, but it’s a function – so you have to call it using “()”.

const count = signal(0);

console.log('The count is: ' + count()); //The count is: 0

**computed(() => signal);**

hasInnerEndOfShelf = computed(() =>

  !!this.listItemData().group?.some(item => item.isEndOfShelf)

);

*listItemData()* is signal - reactive getter function.

Whenever *listItemData()* changes:

* + Angular re-runs the arrow function inside computed()
  + Updates the *hasInnerEndOfShelf* signal’s value

**effect(() => { signals & code });**

effect(

  () => {

    const items = this.filteredDataWithSeparators();

    if (items?.length && !this.isLoading() && this.scrollFacade.currentIndex() < 0) {

      this.scrollFacade.setHighlightingStatusToFirst();

      this.scrollFacade.scrollToHighlighted();

    }

  },

  { allowSignalWrites: true }

);

*filteredDataWithSeparators*, *this.isLoading*, *this.scrollFacade.currentIndex()* are signals, if any of them change – the code within effect will be executed.

Normally, effect() is **read-only** for signals — it shouldn’t modify them, because that could cause infinite loops.

Here, one of methods within effect also updates the signal, the { allowSignalWrites: true } tells Angular: "Yes, I know I'm writing to signals inside this effect — it's intentional."

Because effect() automatically runs whenever dependencies change — no manual subscription or ngOnChanges() needed.

* + Use signal() for local, reactive component state.
  + Use output() for child-to-parent communication (replaces @Output() + EventEmitter).
  + Use Observable for async data streams (like HTTP calls or external data).

## HTML

Angular does not support <script> element in templates.

### Template syntax

@If @else

@if(isServerRunning){

  <span>Yes, the server is running</span>

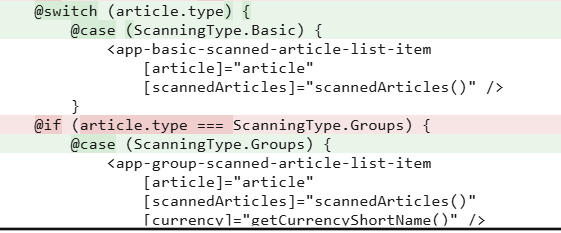
}

@else{

  <span>No, the server is not running</span>

}

isServerRunning may be common variable (not sygnal)

@switch

### Dynamic properties and attributes

You can set:

* + Properties of DOM elements (e.g. [hidden], [disabled], [value])
  + Directives (e.g. \*ngIf, [ngClass], [ngStyle])

But not html attributes like:

<div [visibility]="divVisibility">

 Instead of this use:

You can attach css class depending on bool value:

[class.accept-button-is-editing-position]="isEditingPosition()"

### Event handling (click)=onClicked()

There are many events like (click), (mouseover), (mouseleave), (keyup) etc.

Use (click)=”methodClick()” instead of onclick=”methodClick()”. Angular doesn’t recognize onclick in reactive system;

As I see Angular updates properties in real time even if they are not signals and for both sync and async methods bound to event.

### Input/output data to component

You can create component and insert values in parent component html:

export class User {

  name = input<string>();

  surname = input<string>();

  <app-user name="John" surname="Walker"/>

Or

  @for (user of users; track user.id){

    <app-user

    [name]="user.name"

    [surname]="user.surname"

    />

  }

Also, you can send data back to parent component from child component through defining custom (event) in child’s component:

export class User {

  cardColor = signal('grey');

  addUserNoteEvent = output<string>();

  addItem() {

    this.addUserNoteEvent.emit('User save card color: ' + this.cardColor());

  }

In parent component:

  <app-user (addUserNoteEvent)="addNote($event)"/>

It intercepts the emited by child value and apply as argument to addNote method.

### Deferable views

@defer allows to load some components only when the Event loop is free:

@defer (on viewport) {

  <comments />

} @placeholder {

  <p>Future comments</p>

} @loading (minimum 2s) {

  <p>Loading comments...</p>

}

(on viewport) – optional, it sets the viewport trigger – load only when the user scrolls on this component

@placeholder {

  <p>Future comments</p>

}

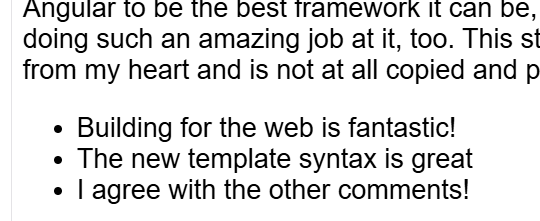
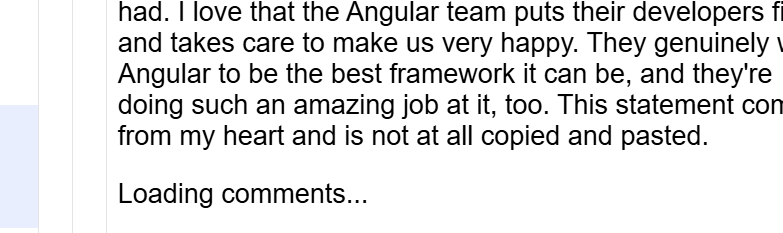
Will be show immediately before the component loading starts (before you even scroll to this place)

@loading (minimum 2s) {

  <p>Loading comments...</p>

}

Will be shown when you scroll to component and during the component loading. (minimum 2s) is optional and enforce to load at least 2 second)



### Pipe | //TO DO

Using “keyValue | pipeName” you can place *keyValue* converted through *pipeName* in html:

    <h2>{{ 'SCANNING.BASIC\_WITH\_PRICE.HEADER' | translate }}</h2>

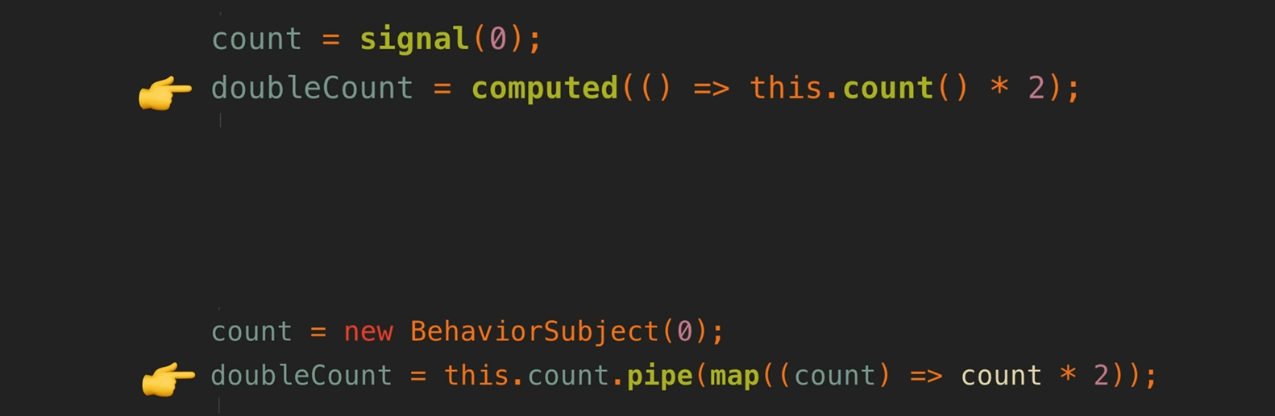
## RxJS

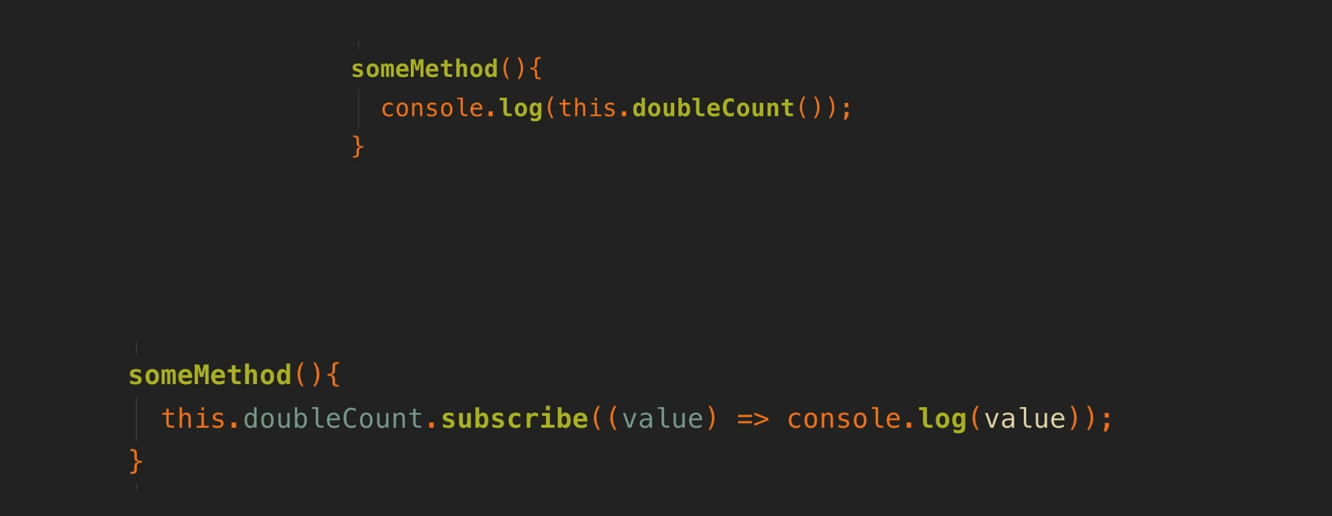
**RxJS** - is a library for reactive programming using Observables, often used in Angular projects.

In old projects the RxJS had two main roles:

* + Coordination async events dealing with race conditions and coordinate async dataflows.
  + Basic reactive building block.

In Angular 17 the Signals were provided. They came to replace RxJS second role – basic reactive building block.

“count = signal(0);” replace old rxjs “count = new BehaviorSubject(0);” (signal is a function, BehaviorSubject – an object)

The problem of rxjs is that doubleCount is now Observable with own rules and you have to subscribe to it, while the doubleCount from signal is a signal:  
and also you must provide unsubscribe strategy (destroy) to avoid unintentional behavior or memory leaks:

But it’s not a big problem since you can import and inject DestroyRef

import { DestroyRef } from '@angular/core';

export class ImsIonInputDirective implements AfterViewInit, OnDestroy {

  private destroyRef = inject(DestroyRef);

And just add

      .pipe(takeUntilDestroyed(this.destroyRef))

To every observable that must be destroyed with destroying the component/service/directive etc. in which it placed.

### Subscribe on observable and effect()

subscribe() in RxJS is like an effect() in Angular Signals, but only on one variable and have to be unsubscribed manually.

this.activatedRouter.queryParams.subscribe((params) => {

  // this function runs whenever the queryParams observable \*emits\* a new value

});

It uses *observable<T>* instead of *signal<T>*

Example:

this.activatedRouter.queryParams.subscribe((params) => {

    const eanFromNav = params['ean'];

    const disableEan = params['disableEan'] === 'true';

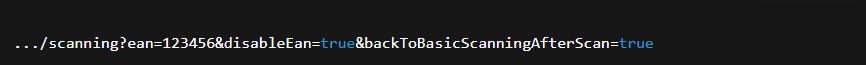
    const backToBasicScanningAfterScan = params['backToBasicScanningAfterScan'] === 'true';

});

*activatedRouter* is an instance of Angular’s *ActivatedRoute*.

*.queryParams* is an observable that emits any time the URL query parameters change.

The observable “emits” mean it produces a new value and sends it to anyone who's subscribed.

*subscribe(...)* lets you react to these changes — e.g. when someone visits:

Angular will emit:

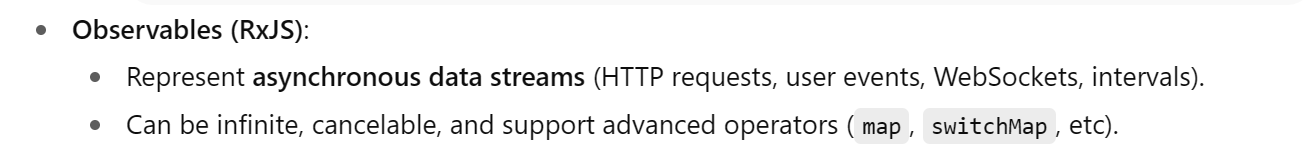
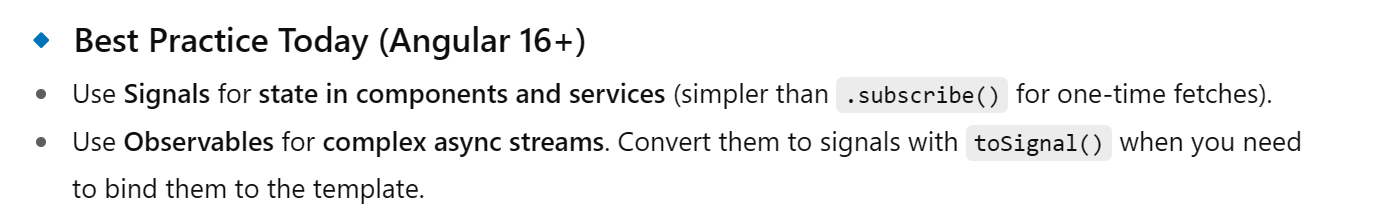
{

  "ean": "123456",

  "disableEan": "true",

  "backToBasicScanningAfterScan": "true"

}



---------------------

    this.settingsMonitorService

      .watch$<UnknownArticleSettings>(this.UNKNOWN\_ARTICLE\_SETTINGS\_KEY)

      .pipe(skip(1), takeUntilDestroyed(this.destroyRef))

      .subscribe((settings) => {

        this.cachedSettings = settings ?? DEFAULT\_UNKNOWN\_ARTICLE\_MODAL\_SETTINGS;

      });

.watch$ is a method of settingsMonitorService ($ points that it returns observable):

    watch$<T = any>(key: string): Observable<T> {

        return new Observable<T>((subscriber) => {

*.pipe(skip(1))*

* + Skips the first emission.
  + Why? Possibly because the initial value is already loaded elsewhere (e.g., via this.loadInitialSettings()).

*takeUntilDestroyed(this.destroyRef)*

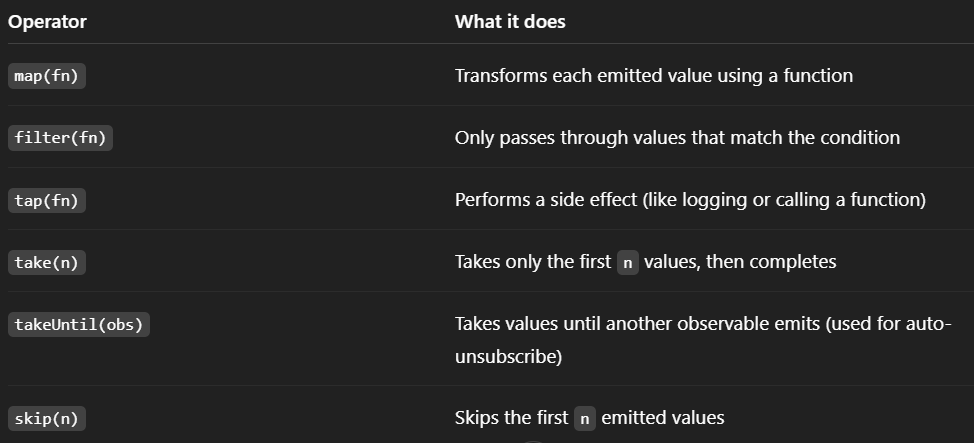
* + Automatically unsubscribes from the observable when the component is destroyed.
  + Cleaner than manually tracking a destroy$ subject.

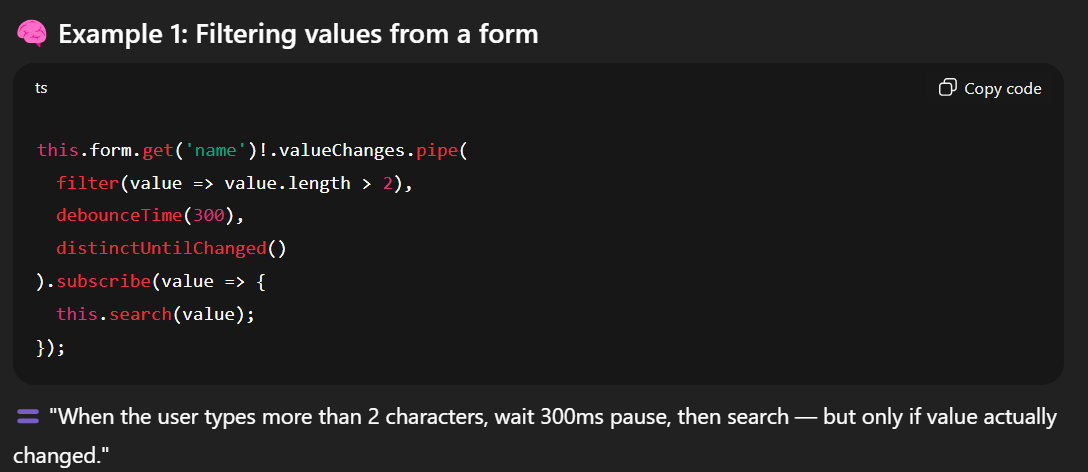
*.subscribe(...)*

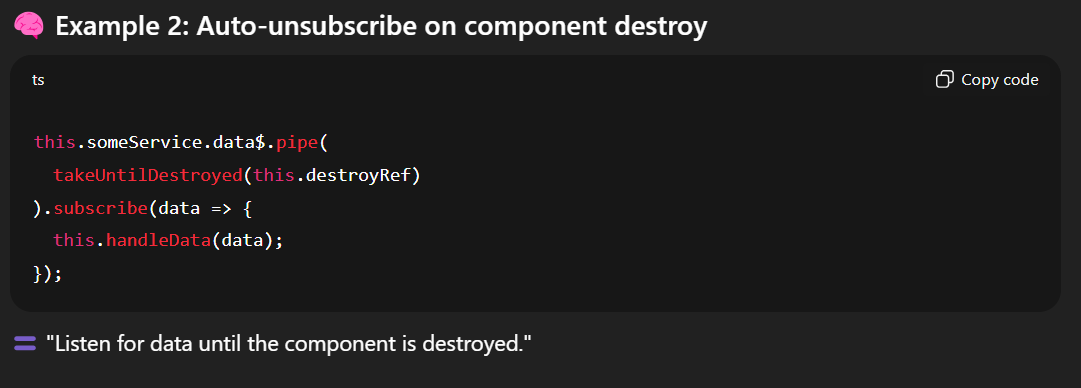
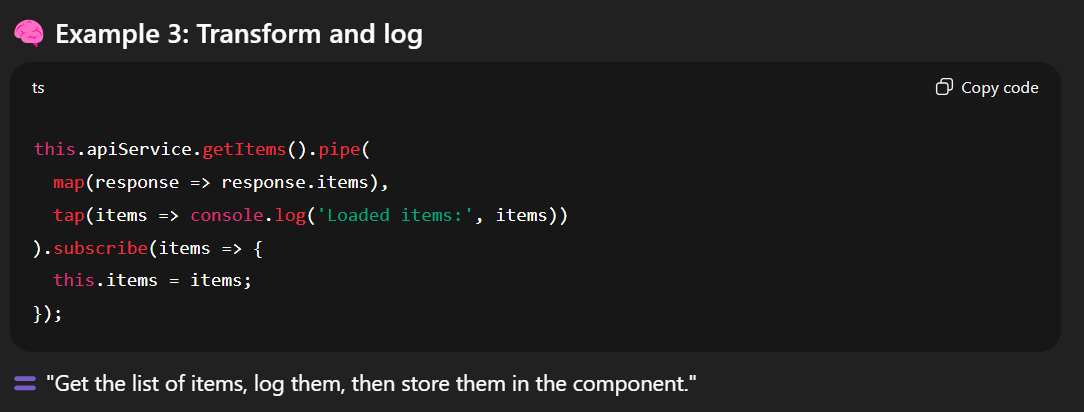
* + Reacts to each emitted value (after skipping the first).
  + Updates the local this.cachedSettings

### .Pipe

In RxJS, *.pipe()* is used to compose a chain of operators that transform, filter, or otherwise handle values emitted by an Observable.

*.pipe* works with operator to define a chain, the examples of operators:

Examples:

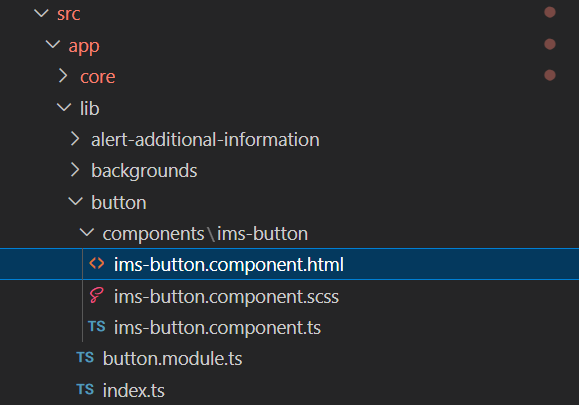


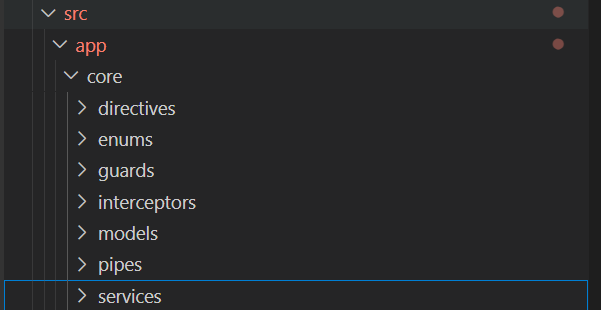
Why use .pipe() instead of chaining?

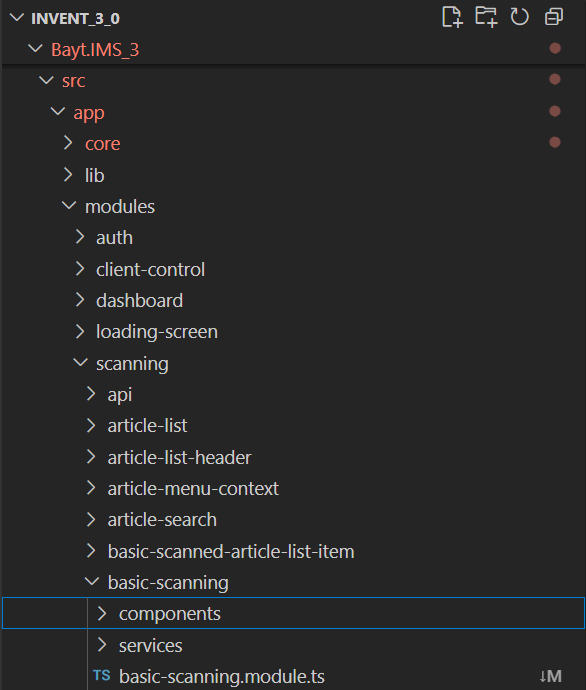
In RxJS v6+, the old way of chaining like observable.map(...).filter(...) is deprecated.

Now, all operators are used inside .pipe(), making the code clearer and more modular (trust me – it’s just better).

## Project architecture example

“Atoms” that used by whole project are in lib directory: src/app/**lib**

src/app/**core** – check the content at picture below. Here are app-scope-available elements. Most services in **core** are injectable “root” (whole app).

most logic is in src/app/**modules**

> auth

> client-control

Are modules. Every module can have nested modules.