by AndЯas

Why typescript?

Typescript (TS) is a superset of JavaScript (JS) that enhances its functionality. It includes a type system that helps catch errors during compilation rather than at runtime. Features like autocomplete and better maintainability (e.g., for tasks like renaming variables or functions) make it a powerful tool for development. While every JavaScript application is compatible with Typescript, the reverse is not true.

In typescript variables are static, in JavaScript its dynamic.

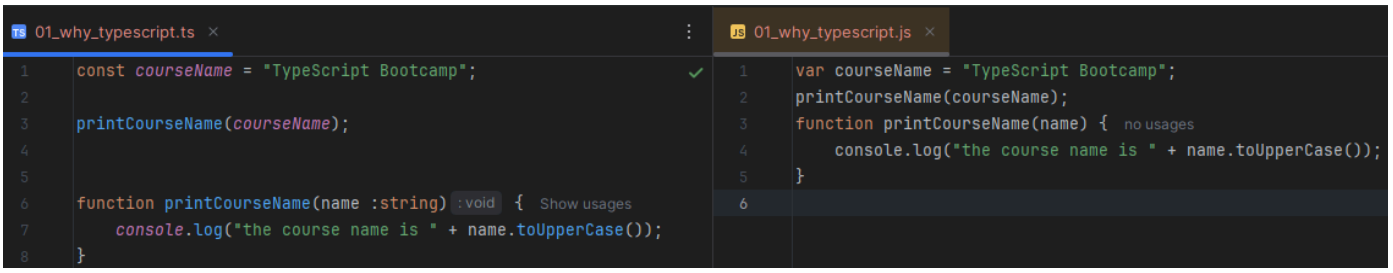
01_why_typescript.js

```
const name = "TypeScript Bootcamp";

printCourseName(name);

printCourseName(100)

printCourseName([1,2,3,])

function printCourseName(name) {  no usages
    console.log("the course name is " + name.toUpperCase());
}

// will come back w TypeError on as they are not strings
// printCourseName(100)
// printCourseName([1,2,3,])
// this will happen in runtime, for example in the browser will come up with the same error
// typescript will help prevent this to happen on the go in the compilation time
```

Typescript compiler

$ npm i -g typescript
$ tsc 01_why_typescript.ts

It transforms a plain JS file next to the TS file. It outputs a JS file that can be run in Node or browser.



01_why_typescript.ts

```
const courseName = "TypeScript Bootcamp";

printCourseName(courseName);


function printCourseName(name :string) :void {  Show usages
    console.log("the course name is " + name.toUpperCase());
}
```

01_why_typescript.js

```
var courseName = "TypeScript Bootcamp";
printCourseName(courseName);
function printCourseName(name) {  no usages
    console.log("the course name is " + name.toUpperCase());
}
```

noEmitOnError Flag

If there is an error in the TS file and we run tsc, it will flag the error, but also will output the JS file. We can prevent this from happening with this flag.
$ tsc --noEmitOnError 01_why_typescript.ts

run in browser
$ npm init
$ npm I lite-server

in packadge.json

```
"scripts": {
  "start": "lite-server"
},
```

create an index.html and pass our JS file

```
</body>

<script src="01_why_typescript.js"></script>
</html>
```

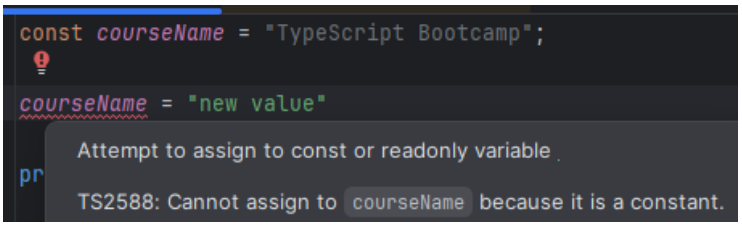
$ npm start

lite-server automatically will do hot reload if we do tsc 01_why_typescript.ts again, but if we would want to we could tell TS to watch:

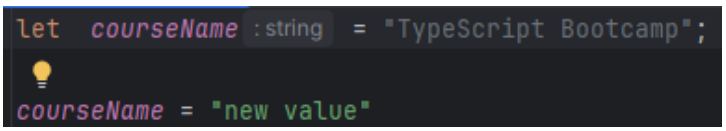
$ tsc --watch --noEmitOnError 01_why_typescript.js

const, let, var

const won`t let you mutate - The value of a constant can't be changed through reassignment using the assignment operator, but if a constant is an object, its properties can be added, updated, or removed.



let – if you need to mutate it over time



var – can be used, but compiler will suggest use let instead the reason being const and let are scoped, but var is global for example here const and let only will be available within the if statement but not after. The only difference let can receive a new value within the statement.

```
if (courseName) {

    const subtitle = "Learn the language fundamentals, build practical projects";

    printCourseName(courseName);
}

console.log(subtitle);
```

```
if (courseName) {

    let subtitle = "Learn the language fundamentals, build practical projects";

    printCourseName(courseName);
}

console.log(subtitle);
```

primitive types – number, string, boolean

```ts
// primitive types: number
const lessonsCount = 10;

const total = lessonsCount + 10;

console.log("total =", total);
```

```ts
// primitive types: string
let title = "Typescript Bootcamp";

let subtitle = "Learn the language fundamentals

const fullTitle = title + ": " + subtitle;

console.log("Full title: " + fullTitle);
```

```ts
// primitive types: boolean
const published = true;

if (published) {
    console.log("The course is published.");
}
```

primitive types: object

```ts
const course = {
    title: "TS bootcamp",
    subtitle: "Listen, look, learn and listen",
    lessonCount: 666
}
```

```ts
const course: {
    title: string
    subtitle: string
    lessonCount: number
}
```

nested objects

```ts
const course = {
    title: "TS bootcamp",
    subtitle: "Listen, look, learn and listen",
    lessonCount: 666,
    author: {
        firstName: "Andras",
        lastName: "Nice",
    }
}
```

```ts
const course: {
    title: string
    subtitle: string
    lessonCount: number
    author: {
        firstName: string
        lastName: string
    }
}
```

template strings

Both " and ' are equivalent, but when you use ` in Ts (and many newer version of JS) other that's a different story.

Template literal strings, if you use const/let use " or ' throughout the program.
Inside template strings, we can also add variables:

```ts
//primitive types: string
const title = "TS";

const subtitle = "TypeScript Bootcamp";

const fullTitle : string  = title + ": " + subtitle;

console.log("full title: ", fullTitle);

const fullTitleTemplateString = `full title: ${title}: ${subtitle}`;

console.log(fullTitleTemplateString)
```

type inference

Type inference in Typescript is the mechanism by which the compiler automatically deduces the type of a variable, function, or expression based on its value or usage, without the need for explicit type annotations.

Key Aspects of Type Inference

1. Variable Initialization: TypeScript infers the type from the initial value assigned to a variable.

```
let message = "Hello, world!"; // Inferred as `string`
let count = 42;               // Inferred as `number`
```

2. Function Return Types: If you omit the return type of a function, TypeScript infers it based on the return statements.

```
function add(a: number, b: number) {
    return a + b; // Inferred as `number`
}
```

3. Contextual Typing: When TypeScript can infer a type based on where a variable or function is used, it's called contextual typing.

```
const handleClick = (event: MouseEvent) => {
    console.log(event.clientX); // `event` is inferred as `MouseEvent` from the context
};
```

4. Default Parameters and Return Types: Default values in function parameters help TypeScript infer their types.

```
function greet(name = "Guest") {
    return `Hello, ${name}`; // Inferred as `string`
}
```

5. Array and Object Literals: The type of array and object literals can also be inferred based on their contents.

```
let numbers = [1, 2, 3]; // Inferred as `number[]`
let person = { name: "Andras", age: 30 }; // Inferred as `{ name: string; age: number }`
```

- Less Boilerplate: Reduces the need to manually write type annotations.
- Improved Readability: Keeps the code clean and concise.
- Type Safety: Ensures that Typescript still checks for type errors based on the inferred type.

Limitations

In some complex scenarios, Typescript may infer a type that's too generic or not what you intended. You can override it using explicit type annotations:

```
let value: string | number = "hello"; // Explicitly specifying a union type
```

Type inference is one of the features that makes Typescript powerful, allowing it to provide a balance between flexibility and strict type checking.

You pretty much can write the whole project without type annotations, the only place it is definitely needed is in-out arguments of functions.

null vs undefined

Variable declared, but not initialized / undefined. It would come us false in runtime

```
let title:string;

console.log("title " + title);
```

```
title undefined
```

```
let title:string;

console.log("title = " + title);

if (!title) {
    console.log("The value of title not known yet.");
}
```

If you have a variable that you know is not yet assigned but you will need it later on, assign it to be null. Null would also come us false in runtime

```
let subTitle:string = null;

console.log("title " + subTitle);
```

```
title null
```

```
let title:string = null;

console.log("title = " + title);

if (!title) {
    console.log("The value of title is not known yet.");
}
```

In TypeScript (and JavaScript), null and undefined are two distinct concepts used to represent the absence of a value.

- Represents intentional absence of a value.
- It is explicitly assigned to a variable to indicate "no value."
- TypeScript treats null as a type, but only if strictNullChecks is off or explicitly allowed.

```
let c: undefined = undefined; // Explicitly assigning undefined
let d: string | undefined;    // Variable can be a string or undefined

console.log(d); // undefined (since no value has been assigned)
```

Key Differences

Aspect	`null`	`undefined`
Meaning	Intentional absence of value	Value not assigned or missing
Type	`null`	`undefined`
Default Value	Not a default value	Default value for uninitialized variables
Explicit Assignment	Needs to be explicitly assigned	Can be implicitly or explicitly assigned
Returned By	Explicitly assigned	Functions without a return value or missing object properties

null to signal that a value is missing but expected, such as in database records or optional fields. undefined is typically used to represent uninitialized states or missing properties in objects.

optional chaining(?)

What if the course is not yet known, it's not ready, in that case it would break and come up as type error cant read null JS, or in TS undefined. To prevent this we can check if it exist first:

```ts
let course : { title: string } = {
    title: "TS Course",
}

if (course.title) {
    console.log(`The title is ${course.title}`);
}
```

```ts
let course : any = null

if (course && course.title) {
    console.log(`The title is ${course.title}`);
}
```

The issue is but what if we have a multiple level of object, there would be a lot of conditions in the if statement and wouldn't be maintainable. To simplify TS has this operator, the only try to access it if its true after '?' :

```ts
let course : any = null

if (course?.textFields?.title) {
    console.log(`The title is ${course.textFields.title}`);
}
```

Optional chaining never meant to replace proper error handling and don't just try to replace . with ?.

```ts
logCourseTitle(course);


function logCourseTitle(course) {

    if (!course?.textFields) {
        console.log("textFields not defined.");
        return;
    }

    if (course.textFields.title) {
        console.log(`The title is ${course.textFields.title}`);
    }
}
```

null coalescing operator(??)

Used to provide a default value when dealing with null or undefined. It is a concise way to handle fallback logic for potentially null or undefined values without triggering on falsy values like 0, false, or an empty string.

You should use this not like in JS || used.

```ts
let course2 : any = null

const title2 : any = course?.textFields?.title ?? "No title available";

console.log(title2);
```

Difference from || (logical OR):

- The || operator considers all falsy values (0, NaN, false, "", null, and undefined) and returns the second operand if the first is falsy.
- The ?? operator only considers null and undefined as invalid.

Common Use Cases:

- Assigning default values for optional fields.
- Providing safe fallbacks when accessing optional properties or results from APIs.

enums

In TypeScript, enums (short for enumerations) are a way to define a set of named constants. They allow you to define a collection of related values that can be numeric or string-based and give those values meaningful names. Enums are especially useful when you want to represent a set of distinct options or categories in a type-safe way.

```ts
enum courseType {  Show usages
    FREE=5,
    PREMIUM =4123 ,
    PRIVATE = "hello"
}

const course = {
    title: 'Ts Course',
    type: courseType.PREMIUM,
}

console.log(course)
```

```ts
enum Direction {
    Up,    // 0
    Down,  // 1
    Left,  // 2
    Right  // 3
}

console.log(Direction.Up);      // Output: 0
console.log(Direction[0]);      // Output: 'Up'
```

```ts
enum Status {
    Success = 200,
    NotFound = 404,
    ServerError = 500
}

console.log(Status.Success);    // Output: 200
console.log(Status[404]);       // Output: 'NotFound'
```

Why Use Enums?

- Readability: Makes your code more self-explanatory by replacing magic numbers or hardcoded strings.
- Type Safety: Ensures only valid options are allowed in your code.
- Reverse Mapping: Numeric enums provide reverse mapping (from value to name).

```ts
enum UserRole {
    Admin = "ADMIN",
    User = "USER",
    Guest = "GUEST"
}

function getPermissions(role: UserRole) {
    switch (role) {
        case UserRole.Admin:
            return "Full Access";
        case UserRole.User:
            return "Limited Access";
        case UserRole.Guest:
            return "Read Only";
    }
}

console.log(getPermissions(UserRole.Admin)); // Output: 'Full Access'
```

any-type

We need to prevent wherever its possible to explicitly declare any, as we wont be type-safe any more, you could assign to anything.

```
let lessionsCount:any = 10;

lessionsCount = "Some string"

 let number:any [] = [10, "anything", true]
```

noImplicitAny

As the compiler here doesn't have any information on title it will implicitly will assign it to any. The lesson count was declared any, the result the same both cases inferred to any.

```
let lessionsCount:any = 10;

lessionsCount = "Some string"

 let number:any [] = [10, "anything", true]

function  printCourse(title :any , lessionsCount :any ) :void  {  no usages
  console.log(`title = ${title}, lessionsCount = ${lessionsCount}`);
}
```

For this reason we always should turn on the noImplicitAny.

$ tsc --watch --noImplicitAny 06_any_type.ts // will flag the issue

union-types / nullable variables

Union types in TypeScript are a powerful feature that allows a variable to hold values of multiple types. They are defined using the vertical bar (|) to separate the types.

```
let value: string | number;

value = "Hello"; // valid
value = 42;      // valid
value = true;    // Error: Type 'boolean' is not assignable to type 'string | number'.
```

Union types are widely used in TypeScript for error handling, function arguments, and defining dynamic data structures.

- Flexibility: Variables can store values of different types without losing type safety.
- Type Guards: Ensures type correctness when working with multiple types.
- Compatibility: Useful for working with APIs or libraries that return multiple types.

At times there can be variables that are optional. The standard use case is to use type null for them as such, above number | string just an example:

```
let courseId: number | null = 1000;

courseId = null;
```

non null assertion operator

The compiler wouldn't throw an error in this but if its run with the --strictNullChecks there will be.

```
let courseId: number | null

courseId.toString(courseId)
```

```
error TS2531: Object is possibly 'null'.
```

Here as the courseId never was initialised it can be possibly null. If we are certain it there will be a value to this we can use the non null assertion operator, this was the compiler will ignore this:

```
let courseId: number | null

courseId!.toString(courseId)
```

strict null checks and the strictNullChecks compiler flag

All types are nullable, string number etc. become null and the compiler wont trough error (would work with undefined as well).

```
let courseId: number = 1000;

courseId = null;
```

This can be an issue but we can prevent this to declare it as a union type number | null and running:

$ tsc --watch --strictNullChecks 07_union_type.ts // will flag the issue

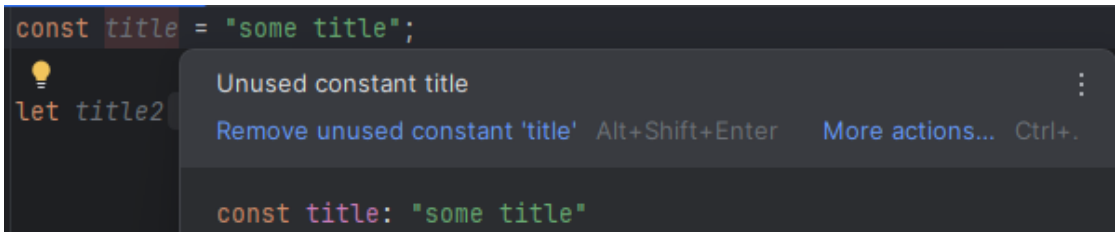
In this case it has to be:

```
let courseId: number | null = 1000;

courseId = null;
```

literal-types

In TypeScript, literal types are specific types that represent exact values rather than broader types like string or number. These can be string literals, numeric literals, or boolean literals, and they are often used to enforce specific allowed values for variables or function parameters.

The compiler assign a type to consts, with let it would assign string:



Defining Allowed Values Literal types are often combined with union types to define a set of allowed values.

```
type Direction = "up" | "down" | "left" | "right";

function move(direction: Direction) {
  console.log(`Moving ${direction}`);
}

move("up");    // Valid
move("back");  // Error: Type '"back"' is not assignable to type '"up" | "down" | "lef
```

type-alias

In TypeScript, a type alias is a way to create a custom name for a type. It's a powerful feature that allows you to define complex or reusable types and give them a meaningful name, making your code cleaner and easier to understand.

We don't want to keep repeating the same union types like courseStatus:"draft" | "unpublished" | ... instead we define a type which we later can reuse.

```
type CourseStatus: "draft" | "unpublished" | "published" | "archived" = "draft";

let courseStatus:CourseStatus = "draft"
```

```
type Course = {
    title:string,
    subtitle:string,
    lessonsCount:number
};

let course: Course = {
    title: "Typescript Bootcamp",
    subtitle: "Learn the language fundamentals, build practical
    lessonsCount: 10
```

- Reusability: Use the same type in multiple places without repeating the definition.
- Readability: Complex types become more understandable with meaningful names.
- Flexibility: Combine primitive types, objects, arrays, and unions into a single alias.

```
type Address = {
  street: string;
  city: string;
  country: string;
};

type UserProfile = {
  id: number;
  name: string;
  address: Address;
};

const profile: UserProfile =
  id: 1,
  name: "Alice",
  address: {
    street: "123 Main St",
    city: "Wonderland",
    country: "Fictionland",
  },
};
```

interfaces

In TypeScript, an interface is a way to define the shape of an object. It specifies the structure that an object must adhere to, including properties and their types. Interfaces are primarily used to enforce type safety and make code easier to understand and maintain.

In this case all properties must be included, if we want to have one of them optional, or just read only so it wont be change, we need to adjust our interface:

```ts
interface  Course {  Show usages
    title: string;
    description: string;
    lessonsCount: number;
}

let course: Course = {
    title: "TS studies",
    description: "Beginner to pro",
    lessonsCount: 12,
}

let otherCourse: Course = {
    title: "Other Ts studies",
    description: "Other pro stuff",
    lessonsCount: 11
}
```

```ts
interface  Course {  Show usages
    readonly title: string;
    description: string;
    lessonsCount: number;
    started?: boolean;
}

let course: Course = {
    title: "TS studies",
    description: "Beginner to pro",
    lessonsCount: 12,
    started: true,
}

let otherCourse: Course = {
    title: "Other Ts studies",
    description: "Other pro stuff",
    lessonsCount: 11
}
```

interfaces vs type aliases

Although they could be used interchangeably, TS recommends:

- Use interfaces when defining object structures or when you expect your type to be extended or merged.
- Use type aliases when dealing with unions, primitives, or more complex type definitions.

Type Alias:

typescript

```ts
type Status = "success" | "failure" | "pending"; // Union
type Coordinates = [number, number]; // Tuple
```

Interface:

typescript

```ts
interface User {
  name: string;
  age: number;
}
```

type assertions

In TypeScript, type assertions are a way to explicitly tell the compiler about the type of a value when TypeScript cannot infer it correctly. Essentially, you're asserting that a value has a specific type, overriding Typescript's inferred type system.

Type assertions do not perform runtime type checks; they only influence the TypeScript compiler and are used purely for static type checking.

In this case the compiler wouldn't know we are trying to get the input field using the DOM.

```
const input = document.getElementById("input-field");
```

So we do this:

```
const input = document.getElementById("input-field") as HTMLInputElement;
```

We overwrite the inferred type that TS would assign.

- Type assertions help when TypeScript cannot infer the correct type or you have more information about the type than TypeScript.
- Use as syntax for consistency and compatibility with JSX.
- Avoid relying too heavily on type assertions to maintain type safety in your code.

module imports/exports

Each TS file is considered as a separate module.

12_modules_inport.ts

```
import {PAGE_SIZE, COURSE, Course} from "./13_modules_exports";

const pageSize :100 = PAGE_SIZE;

const course :{title: string; description: string; lessonCount: number} = COURSE
```

13_modules_exports.ts

```
export type Course = {
    readonly title: string,
    description: string,
    lessonCount?: number,
}

export const PAGE_SIZE = 100  Show usages

export const COURSE = {  Show usages
    title: "TS",
    description: "Bootcamp",
    lessonCount: 10
}
```

re-exports

We can re-export the imported const. Re-exporting in TypeScript is a common practice used to simplify imports and manage modules effectively in larger codebases.

If your project has multiple related modules, you can re-export them from a single file to simplify imports. This is common in feature folders where one index.ts file re-exports all components, utilities, or types.

default exports

In TypeScript (and JavaScript), default exports allow a module to export a single value or entity as the "default" export. This provides a more concise way to export and import a primary item from a module.

When to Use Default Exports

- When a module is designed to export a single, primary value (e.g., a React component, utility function, or class).
- When you want to allow flexibility in naming during imports.

When to Avoid Default Exports

- In modules with multiple exports where clarity and consistency are more important (use named exports instead).
- In large codebases where consistent import names are needed to avoid confusion.

arrow functions

Arrow functions are a concise syntax for writing functions in TypeScript (and JavaScript). They were introduced in ES6 (ECMAScript 2015) and provide a simpler way to write functions, especially for short and inline functions. Arrow functions are particularly useful for improving code readability and avoiding common pitfalls with the this keyword,

```
function saveCourse(course :any , callback: Function) :void {
    setTimeout(function () :void {
        callback("success");
    }, 1000)
}

saveCourse({title: "TS Bootcamp"},
    function() :void {
        console.log("Save was successful!");
    }
)
```

```
function saveCourse(course :any , callback: Function) :void {
    setTimeout(() :void => {
        callback("success");
    }, 1000)
}

saveCourse({title: "TS Bootcamp"},
    () :void =>
        console.log("Save was successful!"))
```

Or this example:

```
const cb = () => console.log("Save successful.");

saveCourse({title:"Typescript Bootcamp"}, cb);
```

But the major difference is that you won't have access to this context if you are using a function as it will create its own context. So for example here the this.course can be accessed in the setTimeout, but if we are using function instead or and arrow functions, it won't know what this.course is. So using the arrow function won`t create a private context.

```
function saveCourse(course, callback: Function) {

    this.course = course;

    setTimeout(() => {

        callback(this.course?.title ?? "unknown course");

    }, 1000);

}

const cb = (title:string) => console.log("Save successful.", title);

saveCourse({title:"Typescript Bootcamp"}, cb);
```

For anonymous functions using an arrow, it's easier to read.

default function arguments

Default function arguments in TypeScript (and JavaScript) allow you to assign default values to parameters in a function. If a parameter is not explicitly provided when the function is called, the default value will be used.

If no argument is passed, the default value ensures the function behaves predictably.

Avoids undefined values in function logic.

```ts
function greet(name: string = 'Guest'): string {
  return `Hello, ${name}!`;
}

// Usage
console.log(greet());          // Output: "Hello, Guest!"
console.log(greet('Andras'));  // Output: "Hello, Andras!"
```

spread operator / shallow vs deep copy

In TypeScript, copying objects can be done either as shallow copy or deep copy, depending on whether you want to copy only the first level of properties or recursively copy all nested objects.

- Shallow Copy is faster and sufficient for flat objects or when nested objects don't need to be independent.
- Deep Copy is necessary for deeply nested objects or when you want a true independent clone.

From this example you can see is course.stats.lessionCount referenced the console.log will give lessonCount as 10 in both cases, but if its referenced as course.stats the first log will be 12 but the second will be 100

```ts
let course: Course = {
    title: "TS",
    subtitle: "Bootcamp",
    stats: {
        lessonCount: 12,
    }
}

const newCourse = {
    title: course.title,
    subtitle: course.subtitle,
    stats: {
        course: course.stats.lessonCount,
        //deep copy if referencing course.stats it would be a shallow copy
    }
}

console.log(newCourse)

course.stats.lessonCount = 100

console.log(newCourse)
```

```
{ title: 'TS', subtitle: 'Bootcamp', stats: { course: 12 } }
{ title: 'TS', subtitle: 'Bootcamp', stats: { course: 12 } }
{ title: 'TS', subtitle: 'Bootcamp', stats: { lessonCount: 100 } }
```

With spread op we are doing a shallow copy

object destructuring

Object destructuring in TypeScript allows you to extract properties from an object and assign them to variables, simplifying code and improving readability. TypeScript enhances this feature by ensuring type safety and enabling the use of interfaces and types.

```
let course: Course = {
    title: "TS",
    subtitle: "Bootcamp",
    lessonCount: 12,
}

function printCourse(course: Course) {  no usages
    console.log(`title: ${course.title}, subtitle: ${course.subtitle} lessonCount: ${course.lessonCount}`);
}
```

```
printCourse(course);

function printCourse(course: Course) {  Show usages

    const {title, subtitle, lessonCount} = course;

    console.log(`title: ${title}, subtitle: ${subtitle} lessonCount: ${lessonCount}`);
}
```

or you could use spread op, if you don't need all the values:

```
function printCourse(course:Course) {

    const {title, ...rest} = course;
```

good example:

```
const user = { name: "Andras", age: 30, city: "London" };

const { name, age } = user;

console.log(name); // "Andras"
console.log(age);  // 30
```

array spread op destructuring

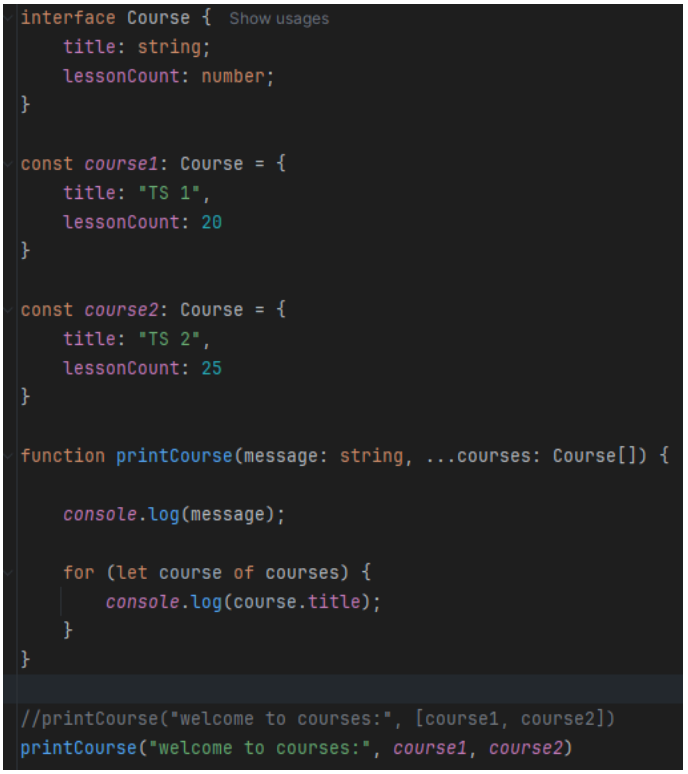
```
const numbers :number[]  = [1,2,3,4,5];

const moreNumbers :number[]  = [...numbers, 6,7,8];

console.log(moreNumbers);

const [first, second, third] = moreNumbers

console.log(first, second, third);
```

```
[
  1, 2, 3, 4,
  5, 6, 7, 8
]
1 2 3
```

rest function arguments

In TypeScript, rest function arguments refer to a way to capture a variable number of arguments passed to a function into a single array. This is done using the rest parameter syntax (...). Rest parameters are particularly useful when working with functions that accept an arbitrary number of arguments.

```ts
interface Course {  Show usages
    title: string;
    lessonCount: number;
}

const course1: Course = {
    title: "TS 1",
    lessonCount: 20
}

const course2: Course = {
    title: "TS 2",
    lessonCount: 25
}

function printCourse(message: string, courses: Course[]) {

    console.log(message);

    for (let course of courses) {
        console.log(course.title);
    }
}

printCourse("welcome to courses:", [course1, course2])
```



```ts
interface Course {  Show usages
    title: string;
    lessonCount: number;
}

const course1: Course = {
    title: "TS 1",
    lessonCount: 20
}

const course2: Course = {
    title: "TS 2",
    lessonCount: 25
}

function printCourse(message: string, ...courses: Course[]) {

    console.log(message);

    for (let course of courses) {
        console.log(course.title);
    }
}

//printCourse("welcome to courses:", [course1, course2])
printCourse("welcome to courses:", course1, course2)
```

Key Points

- A function can have only one rest parameter, and it must appear at the end of the parameter list.
- The rest parameter is strongly typed as an array of the specified type.
- Default values cannot be used directly with rest parameters, but can be simulated by defaulting the array to an empty array.

rest parameters vs spread syntax

- Rest Parameters are used to collect multiple arguments into a single array.
- Spread Syntax is used to expand an array into individual elements.

```
function sum(...numbers: number[]): number {
  return numbers.reduce((acc, curr) => acc + curr, 0);
}

console.log(sum(1, 2, 3, 4)); // 10
console.log(sum(10, 20));     // 30
```

```
function multiply(multiplier: number, ...nums: number[]): number[] {
  return nums.map(num => num * multiplier);
}

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

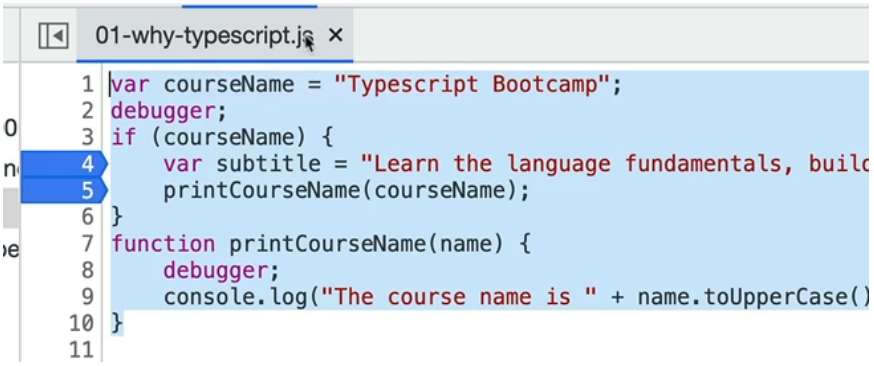
```
function printMessages(...messages: string[]): void {
  console.log(messages.length > 0 ? messages : ["No messages"]);
}

printMessages("Hi", "Hello"); // ["Hi", "Hello"]
printMessages();              // ["No messages"]
```

debugging typescript / source map in browser environment

In a JavaScript file we can add a debugger and add multiple steps in the browser but a larger project transpile to JS will be hard to read, so we are better off debugging in TS. In this example we are using lite-server to run the page for the browser.

```typescript
const courseName = "Typescript Bootcamp";

debugger;

if (courseName) {

    const subtitle = "Learn the language fundamentals, build practical projects";

    printCourseName(courseName);
}

function printCourseName(name :string) {

    debugger;

    console.log("The course name is " + name.toUpperCase());

}
```

$ tsc 01-why-typescript.ts0
$ npm start

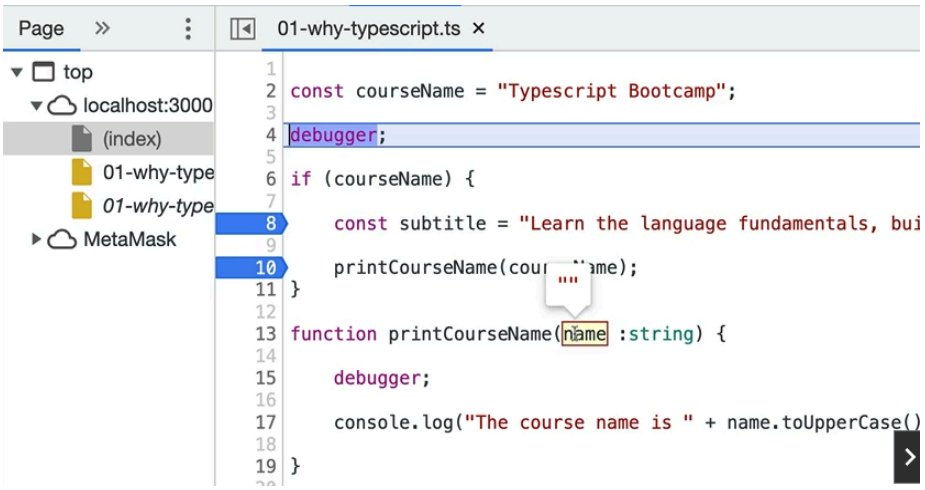



```
"scripts": {
  "start": "lite-server"
},
```

What we can do is to run:

$ tsc –sourceMap 01-why-typescript.ts
$ npm start

This will output apart from the JS file a js.map file and the debugger will be showing the TS file:

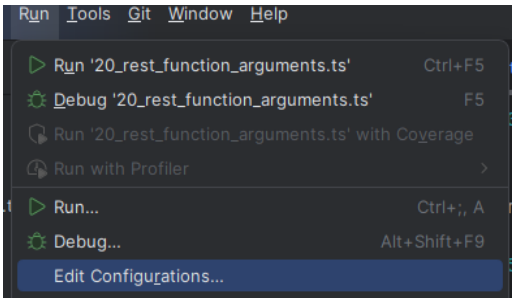


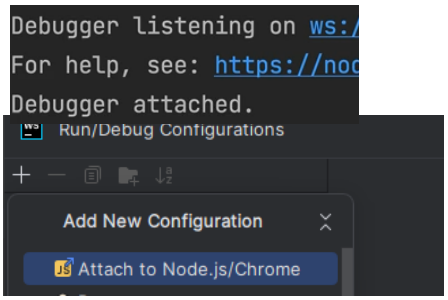
debugging TS in Node / source map

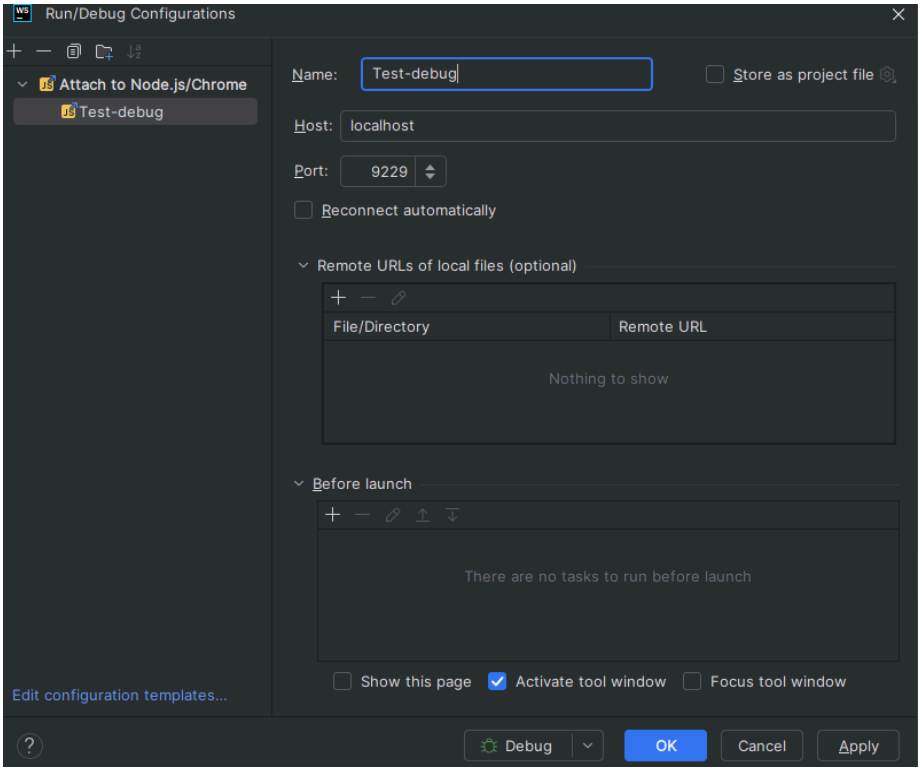
As we are running a simple script it would run without the debugger being attached. In order to be break it up we would use –-inspect-brk –-inspect-brk instead of just --inspect

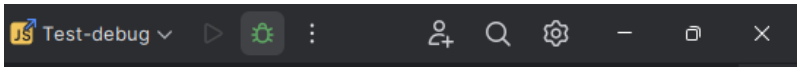
$ tsc –sourceMap 01-why-typescript.ts
$ node –-inspect-brk 01-why-typescript.js

```
Debugger listening on ws://127.0.0.1:9229/54be80df-a5ab-467c-8c17-a7a28e3110dd
For help, see: https://nodejs.org/en/docs/inspector
```










```ts
const courseName = "Typescript Bootcamp";

debugger;

if (courseName) {

    const subtitle = "Learn the language fundamentals, build practical projects";

    printCourseName(courseName);
}

function printCourseName(name :string) {

    debugger;

    console.log("The course name is " + name.toUpperCase());

}
```

shorthand object creation notation

A concise way to create objects in JavaScript, especially when the property names match the variable names.

course_title.ts

```ts
interface Course {  Show usages
    title: string;
    subTitle: string;
    lessonCount: number;
}

const courseTitle = "TS",
 courseSubtitle = "Bootcamp",
 lessonCount = 20

const course: Course = {
    title: courseTitle,
    subTitle: courseSubtitle,
    lessonCount: lessonCount,
}
```

course_title.ts

```ts
interface Course {  Show usages
    title: string;
    subTitle: string;
    lessonCount: number;
}

const title = "TS",
    subTitle = "Bootcamp",
 lessonCount = 20

const course: Course = {
    title,
    subTitle,
    lessonCount
}
```

Example

javascript Copy code

```javascript
const name = "Andras";
const age = 30;

// Shorthand notation
const person = { name, age };

console.log(person);
// Output: { name: 'Andras', age: 30 }
```

Explanation

- Normally, you would write:

  javascript Copy code

  ```javascript
  const person = { name: name, age: age };
  ```

- But with shorthand notation, you can simply write `{ name, age }` when the property name is the same as the variable name.

ts functions

We always need to add the input parameters type annotations of the functions otherwise TS will assignt any. Use explicit types for parameters and return values.

No return in the function not type annotation for functions, showing type void = no return value:

```
interface Course {  no usages
    title: string;
    subTitle: string;
    lessonCount: number;
}

function createCourse(title: string, subTitle: string, lessonCount: number)  {  Show usages
    console.log(`Creating TS course with Title: ${title}, Subtitle: ${subTitle}, lessonCount: ${lessonCount}`);

}

const results : void  = createCourse("TS", "Bootcamp", 20)
```

There is return, but not assigned type to return, inferring the value :

```
interface Course {  no usages
    title: string;
    subTitle: string;
    lessonCount: number;
}

function createCourse(title: string, subTitle: string, lessonCount: number)  {  Show usages
    console.log(`Creating TS course with Title: ${title}, Subtitle: ${subTitle}, lessonCount: ${lessonCount}`);

    return {
        title,
        subTitle,
        lessonCount,
    }
}

const results : { title: string; subTitle: string; lesson...  = createCourse("TS", "Bootcamp", 20)
```

Using Course interface.

```
interface Course {  Show usages
    title: string;
    subTitle: string;
    lessonCount: number;
}

function createCourse(title: string, subTitle: string, lessonCount: number) : Course  {  Show usages
    console.log(`Creating TS course with Title: ${title}, Subtitle: ${subTitle}, lessonCount: ${lessonCount}`);

    return {
        title,
        subTitle,
        lessonCount,
    }
}

const results : Course  = createCourse("TS", "Bootcamp", 20)
```

Or could be do the same adding as:

```
return {
    title,
    subTitle,
    lessonCount,
} as Course;
```

functions

Functions can be used as a type. You can explicitly define the shape of a function using its parameter types and return type. Function is a value at runtime just as a string or a number.
same thing:

```
function createCourse(title:string, subtitle:string,
                      lessonsCount:number) :Course {

    console.log(` Creating course with Title: ${title},
    Subtitle: ${subtitle} lessons count: ${lessonsCount}`);

    return {
        title,
        subtitle,
        lessonsCount
    };
}
```

```
const createCourse = (title:string, subtitle:string,
                      lessonsCount:number) :Course => {

    console.log(` Creating course with Title: ${title},
    Subtitle: ${subtitle} lessons count: ${lessonsCount}`);

    return {
        title,
        subtitle,
        lessonsCount
    };
}
```

tuples

In TypeScript, tuples are a special type of array that allows you to define a fixed-length array with specific types for each element. Unlike regular arrays where all elements are of the same type, tuples allow each position in the array to have a distinct type

- Fixed Length: The number of elements in a tuple is predefined.
- Typed Elements: Each element in the tuple has a specific type, and the order matters.
- Index-Based: You can access elements using their index like regular arrays.

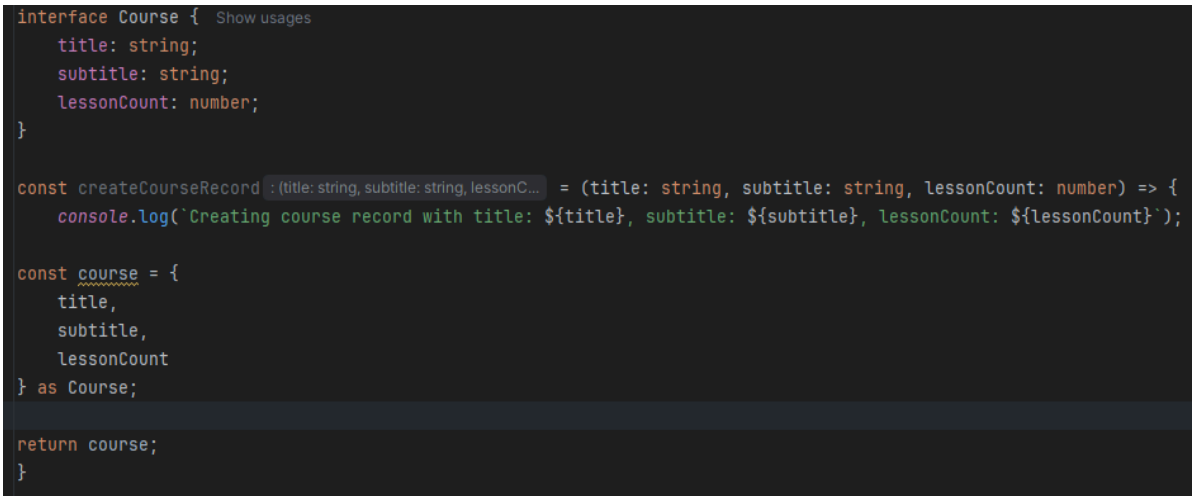
```
let tupleName: [Type1, Type2, ..., TypeN];
```

```
interface Course {  no usages  Andras E
    title: string;
    subTitle: string;
    lessonCount: number;
}

type CourseRecord = [string, string, number]

const courseRecord: CourseRecord = ["TS", "Bootcamp", 20]
```

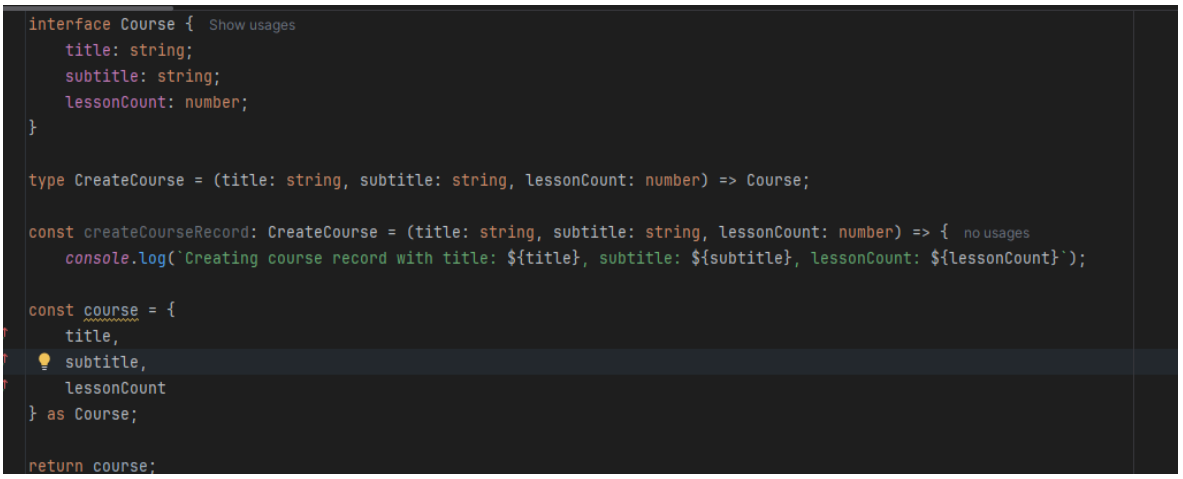
function types

In Typescripts, function types specify the types of arguments a function takes and the type of value it returns. This helps ensure type safety by verifying that functions are called with the correct arguments and return the expected result.

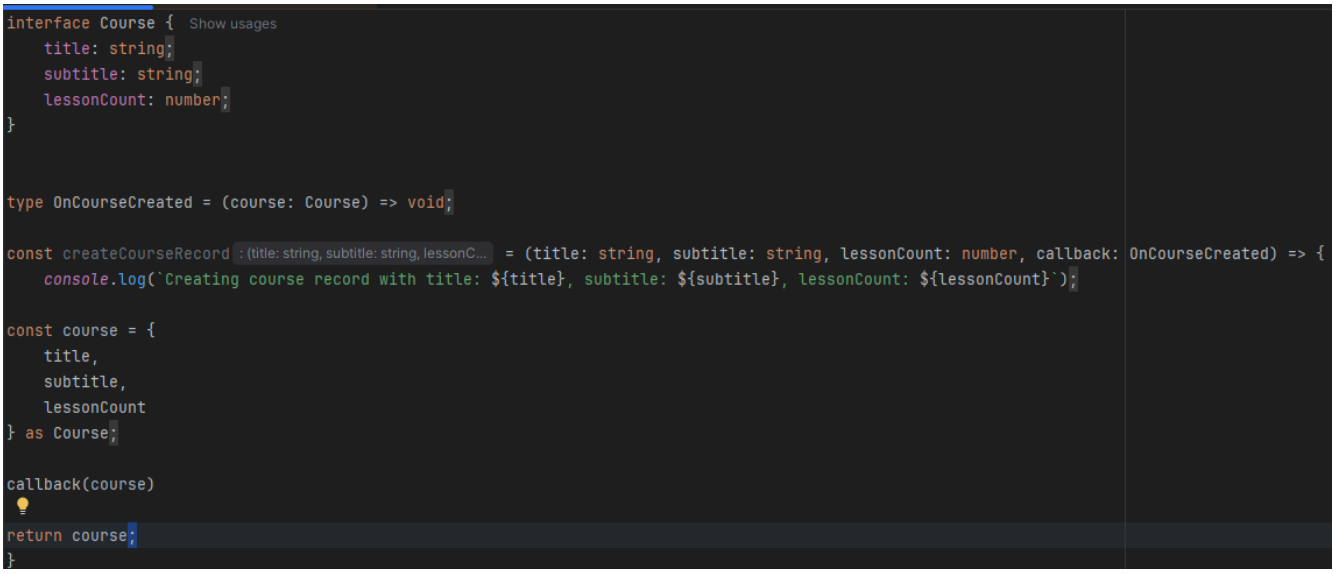
Here we have type inference as we didn't declare it.



Make the function type safe:



Callback:



unknown type

In Typescript, the unknown type represents a value that could be of any type but is safer than the any type. It acts as a type-safe counterpart to any by requiring you to explicitly perform type checks or type assertions before using the value.

- Values of type unknown cannot be used without some form of type checking.
- This ensures that developers handle unknown types explicitly, preventing runtime errors.
- You can assign any value to a variable of type unknown.
- However, the restrictions come into play when you try to use the value. For example, performing operations on it, accessing properties, or calling methods requires type narrowing or assertions.

```
let value: unknown;

// Assignments are always allowed
value = 42;          // OK
value = "hello";     // OK
value = { key: true }; // OK

// But trying to use the value causes errors
console.log(value.toUpperCase()); // Error: Object is of type 'unknown'
console.log(value.key);           // Error: Object is of type 'unknown'
```

Checking the type before (type narrowing)

```
if (typeof unknownValue == "string") {

    let value14: string = unknownValue;
}
```

type narrowing and type predicates

Type predicates in TypeScript are a way to tell the compiler that a value has a specific type, typically as part of a custom type guard. They are used in functions to narrow the type of a variable in a type-safe manner.

Use cases:

- Custom Type Guards: Creating reusable functions for narrowing types.
- Refining Union Types: Distinguishing between multiple possible types in a union.
- Working with unknown: Safely narrowing unknown types to specific ones.

```
function isType(value: unknown): value is Type {
  // Return a boolean indicating whether the value is of the specified type
}
```

```
function isString(value: unknown): value is string {
  return typeof value === "string";
}

let input: unknown = "hello";

if (isString(input)) {
  console.log(input.toUpperCase()); // Safe, input is narrowed to 'string'
}
```

type never

In TypeScript, the never type represents values that never occur. It is the most restrictive type in TypeScript and is typically used in scenarios where a value cannot possibly exist.

- Variables of type never cannot hold any value. No valid value can be assigned to never.
- Functions or code paths that throw errors or never return are inferred to have the never type.
- TS knows that here the else statement will be never as the type is either draft or published.

```
type CourseStatus = "draft" | "published";

let courseStatus: CourseStatus;

if (courseStatus == "published") {
    console.log("published")
} else if (courseStatus == "draft") {
    console.log("draft")
} else {
    const value :never  = courseStatus
}
```

```
let courseStatus : CourseStatus;

if (courseStatus == "draft") {

}
else if (courseStatus == "published") {

}
else {
    unexpectedError(courseStatus);
}

function unexpectedError(value:never) {
    throw new Error(`Unexpected value: ${value}`);
}
```

Common Use Cases

- Functions That Never Return: A function that either throws an error or runs forever can have a return type of never.
- Exhaustive Checks: never is useful in exhaustive type checking for union types. It ensures that all possible cases are handled in a switch statement or similar constructs.
- Unreachable Code: Code that TypeScript determines will never execute can infer the never type.

never vs void:

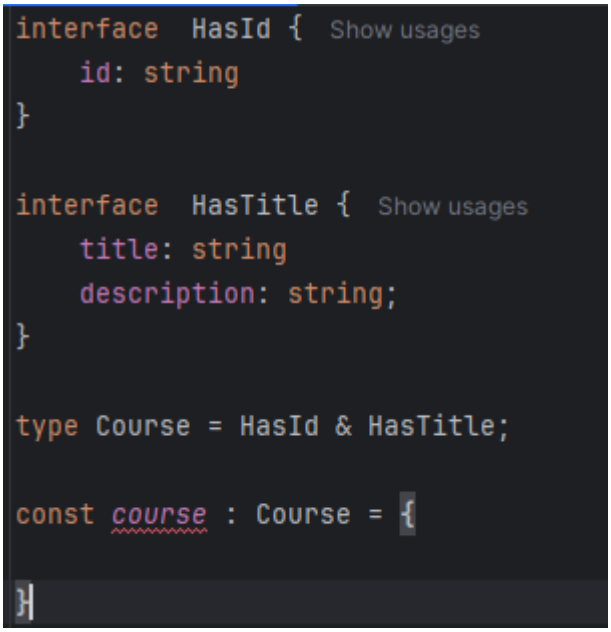
- void is used for functions that return nothing (but do terminate normally).
- never is used for functions that do not return at all (e.g., they throw or loop forever).

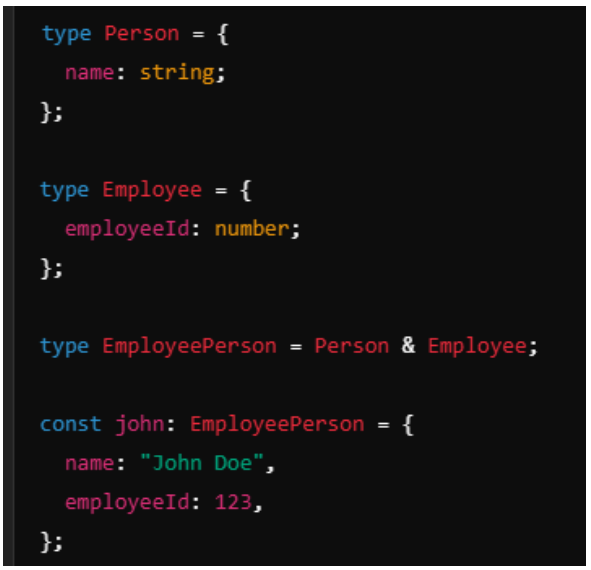
never vs unknown:

- unknown represents a type whose value is not yet known and can hold anything.
- never represents a type that can never have a value.

intersection types

As opposed to union types, an intersection type is a way to combine multiple types into one. A value of an intersection type must satisfy all the combined types. Intersection types are created using the & (ampersand) operator.





tsconfig.json

TypeScript project is a configuration file that specifies the settings for the TypeScript compiler (tsc). It allows developers to define how TypeScript should compile their code, what files to include or exclude, and many other settings that control the behavior of the TypeScript environment.

```json
{
  "compilerOptions": {
    "target": "ES6", // JavaScript version to compile to
    "module": "commonjs", // Module system to use
    "strict": true, // Enable strict type-checking
    "outDir": "./dist", // Output directory for compiled files
    "rootDir": "./src", // Root directory of source files
    "esModuleInterop": true, // Enables ES Module compatibility
    "skipLibCheck": true // Skip type checking for .d.ts files
  },
  "include": ["src/**/*"], // Files or directories to include
  "exclude": ["node_modules", "dist"] // Files or directories to exclude
}
```

It doesn't have to be called tsconfig.json, but when it's called in the terminal we need to use a project flag.

```
fundamentals % tsc --project custom-tsconfig.json
```

We can also overwrite the config file on calling for example:

```
--project custom-tsconfig.json --target es5
```

Run tsc command would compile all the files, but we can use it for a few files as such:

```json
{
  "compilerOptions": {
    "target": "ES5"
  },
  "files": [
    "21_shorthand_object_creation_notation.ts",
    "26_type_predicates.ts"
  ]
}
```

We also can include group of files for example folders, in this case we include all files from src directory including folders and all files using wildcards:

```json
{
  "compilerOptions": {
    "target": "es5"
  },
  "include": [
    "src/**/*"
  ]
}
```

We also can use exclude but its only works in relation to include:

We can also declare a root directory

Output would be the tsconfig`s location, we usually don't want this:

tsconfig.json module

In a tsconfig.json file, the "module" option specifies the module system that TypeScript should use when compiling the code. This option tells TypeScript how to treat import and export statements in the code and how to emit the corresponding output.

"commonjs": This is the default for Node.js applications. It compiles the TypeScript code to CommonJS modules, which are widely used in Node.js. The output uses require() and module.exports.

```
{
  "compilerOptions": {
    "target": "es5",
    "rootDir": "src",
    "outDir": "dist",
    "module": "CommonJS"
  }
}
```

It will compile the TS file to commonJS style syntax with require:

```
"use strict";
Object.defineProperty(exports, "__esModule", { value: true });
var _12_modules_exports_1 = require("./12-modules-exports");
var pageSize = _12_modules_exports_1.PAGE_SIZE;
var _15_default_exports_1 = require("./15-default-exports");
(0, _15_default_exports_1.default)({});
```

There are other options such as: commonjs, esnext, es6, amd, system, umd ...

tsconfig.json lib

In a tsconfig.json file, the "lib" option specifies a list of library files that TypeScript should include during the compilation process. These libraries provide type definitions for various JavaScript and browser features or APIs, such as the DOM, ES modules, or Node.js APIs.

The "lib" option allows you to control which built-in JavaScript features and types are available to your TypeScript code. This can be particularly useful if you are targeting specific environments or versions of JavaScript.

Here for example The lib array includes "dom", "es2015", and "webworker", so TypeScript will include the appropriate type definitions for DOM APIs, ECMAScript 2015 features, and Web Workers.

```
{
  "compilerOptions": {
    "target": "es5",
    "lib": ["dom", "es2015", "webworker"]
  }
}
```

The default options include a good set of libraries, unless we have specific requirements we can stick to that.

tsconfig.json baseUrl

By setting a baseUrl, you can define a root directory for module imports, and TypeScript will resolve module imports relative to this base path. This makes it easier to manage your import paths, especially in larger projects.

```
src/
  components/
    Button.ts
  utils/
    helpers.ts
  app.ts
```

```jsonc
{
  "compilerOptions": {
    "baseUrl": "./src",  // Sets the base directory for module resolution
    "paths": {
      "components/*": ["components/*"],
      "utils/*": ["utils/*"]
    }
  }
}
```

Without baseUrl

```ts
import { Button } from './components/Button';  // Relative path
import { helpers } from './utils/helpers';    // Relative path
```

W baseUrl

```ts
import { Button } from 'components/Button';  // Non-relative import
import { helpers } from 'utils/helpers';    // Non-relative import
```

typeRoots, types (adding extra types)

In TypeScript, the typeRoots and types options in the tsconfig.json file control how TypeScript discovers and includes type definitions from external libraries and custom type declaration files.

The typeRoots option specifies the directories where TypeScript should look for type declaration files (*.d.ts). By default, TypeScript includes the node_modules/@types directory as a place to look for type definitions, but you can customize this behavior using the typeRoots option.

When you specify typeRoots, TypeScript will only look in the provided directories for type declaration files and will not automatically look in node_modules/@types unless you explicitly include that in the paths.

If you do not specify typeRoots, TypeScript will automatically include node_modules/@types for type declarations.

```
{
  "compilerOptions": {
    "typeRoots": ["./types", "./node_modules/@types"]
  }
}
```

For example we install express: npm i express, this package not built originally with ts in order for the compiler to include the benefit we need to install:
npm install -D @types/express the D flag is its only devDependency as it doesn`t add to the project but is used for the benefit of writing code.

```
{
  "name": "fundamentals",
  "version": "1.0.0",
  "description": "",
  "scripts": {
    "start": "../node_modules/.bin/lite-server"
  },
  "author": "",
  "license": "ISC",
  "dependencies": {
    "express": "^4.21.2",
    "lite-server": "^2.6.1"
  },
  "devDependencies": {
    "@types/express": "^5.0.0"
  }
}
```

The types option specifies a list of type definition packages or modules that TypeScript should include. If you use this option, TypeScript will only include the type definitions listed in the array. This allows you to explicitly define which type definitions should be included in the compilation, rather than letting TypeScript automatically include all types from node_modules/@types.

```
{
  "compilerOptions": {
    "types": ["node", "jest"]
  }
}
```

Combining Both typeRoots and types:

- TypeScript will first look in ./customTypes and ./node_modules/@types for type definitions.
- It will only include the types for node and jest from node_modules/@types, ignoring other types from @types.

```
{
  "compilerOptions": {
    "typeRoots": ["./customTypes", "./node_modules/@types"],
    "types": ["node", "jest"]
  }
```

skipLibCheck

The skipLibCheck option in TypeScript is a compiler setting that allows you to skip type checking of declaration files (.d.ts files), which are typically used to provide type definitions for libraries.

When working with large projects or many third-party libraries, TypeScript can spend significant time type-checking the .d.ts files of dependencies. Some libraries might have complex or poorly written type definitions that can result in errors unrelated to your project.

Enabling skipLibCheck improves build performance and avoids type-checking issues in external libraries by skipping the type-checking step for these files.

```
{
  "compilerOptions": {
    "skipLibCheck": true
  }
}
```

Advantages

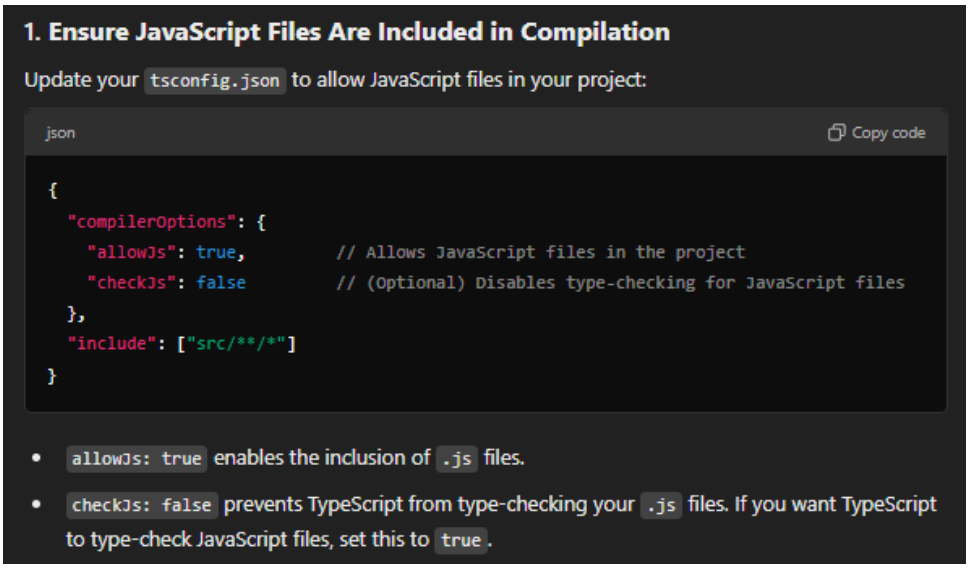
- Improved Compilation Speed: Skipping type checking for library files can significantly speed up compilation, especially in large projects with many dependencies.
- Avoiding Library Type Errors: If a third-party library contains problematic or incorrect type definitions, enabling skipLibCheck prevents those issues from blocking your project.
- Focus on Your Code: Allows you to focus on checking your code while relying on the assumption that the library authors have tested their type definitions.

Disadvantages

- Potential Type Incompatibilities: If a library has incorrect or incompatible types, these issues might propagate to your project without you noticing.
- Less Rigor: Skipping type checks in library files may allow subtle type errors to slip through.

```
// Example Error
node_modules/some-library/index.d.ts:15:5 - error TS2322: Type 'number' is not assignable to type 'string'.
```

allowJS (plain javascript in a TS project)



1. Ensure JavaScript Files Are Included in Compilation

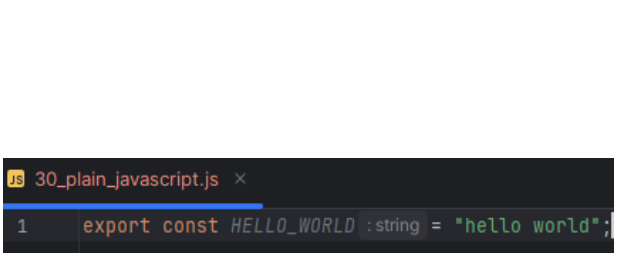
Update your `tsconfig.json` to allow JavaScript files in your project:

json — Copy code

```
{
  "compilerOptions": {
    "allowJs": true,         // Allows JavaScript files in the project
    "checkJs": false         // (Optional) Disables type-checking for JavaScript files
  },
  "include": ["src/**/*"]
}
```

- `allowJs: true` enables the inclusion of `.js` files.
- `checkJs: false` prevents TypeScript from type-checking your `.js` files. If you want TypeScript to type-check JavaScript files, set this to `true`.

with this we now have auto compilation:

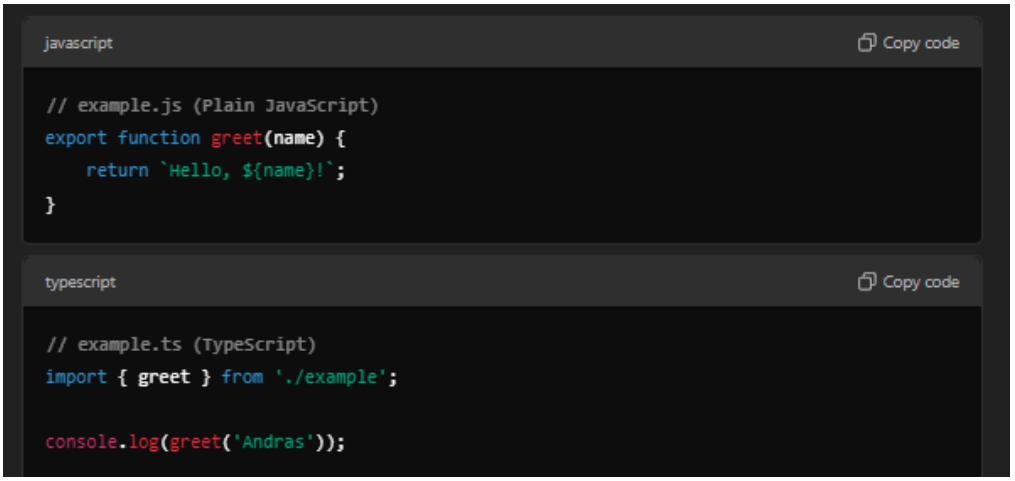


30_plain_javascript.js

```
1   export const HELLO_WORLD :string = "hello world";
```

```
const express :any  = require('express');

import {Request, Respond} from "express";
import {HELLO_WORLD} from "./30_plain_javascript";

let request: Request;
let response: Response;
```

referencing JS in TS



javascript — Copy code

```
// example.js (Plain JavaScript)
export function greet(name) {
    return `Hello, ${name}!`;
}
```

typescript — Copy code

```
// example.ts (TypeScript)
import { greet } from './example';

console.log(greet('Andras'));
```

To have the TS benefit and have type checking we need to have checkJS set to be true. By default it would be false.

Provide Type Declarations (Optional but Recommended)

To benefit from TypeScript's type-checking and IntelliSense, create a declaration file (.d.ts) for the JavaScript file. The other option is to Provide Type Declarations (Optional but Recommended). We benefit from TypeScript's type-checking and IntelliSense by creating a declaration file (.d.ts) for the JavaScript file.

```
// example.d.ts
export function greet(name: string): string;
```

miscellaneous options

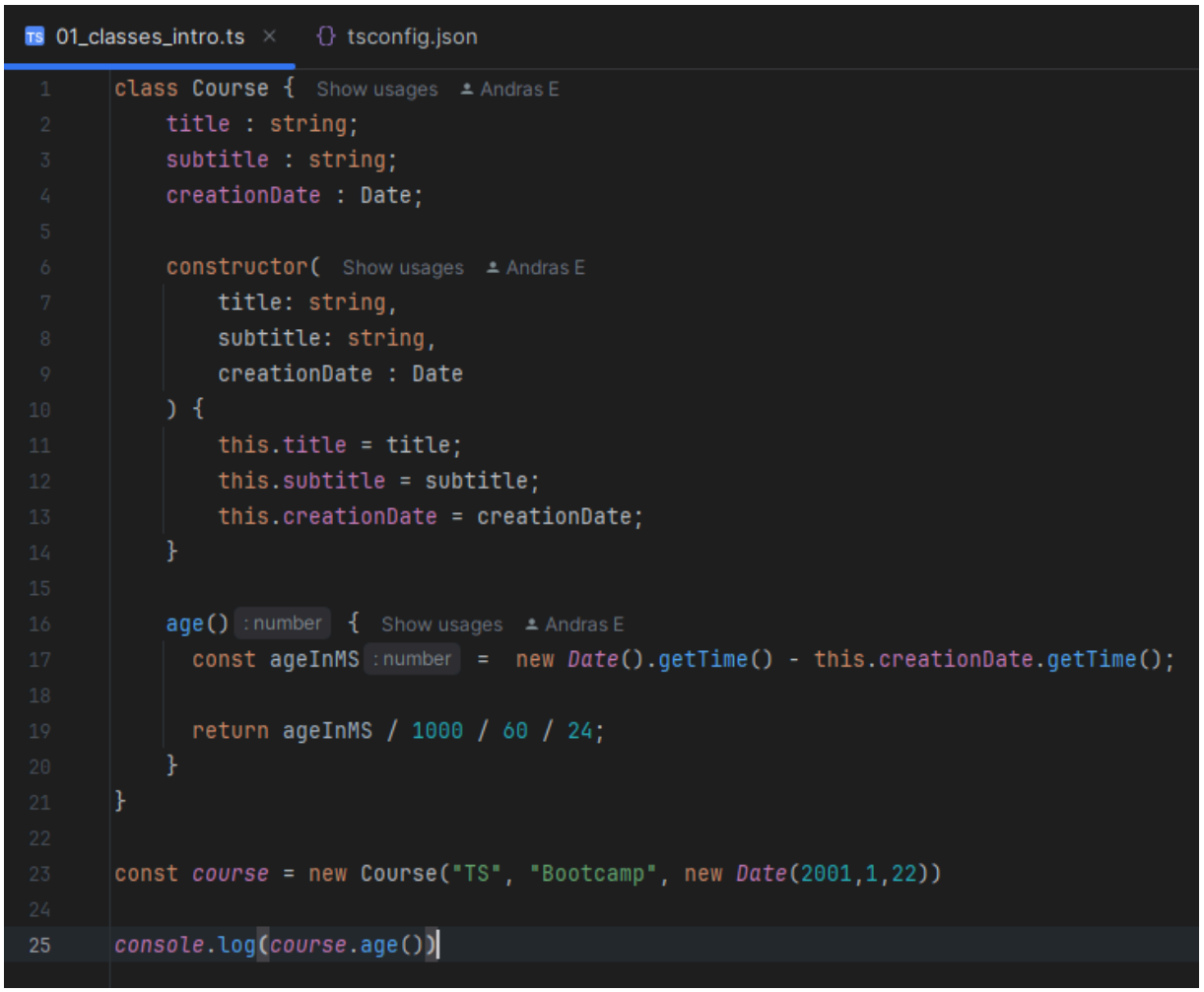
- allowUnusedLabels: Allows unused labels in JavaScript and TypeScript code.
- allowUnreachableCode: Prevents errors for unreachable code.
- noErrorTruncation: Prevents truncation of error messages in the terminal or console.
- suppressExcessPropertyErrors: Suppresses errors for extra properties in object literals during type checking.
- suppressImplicitAnyIndexErrors: Suppresses errors for using an implicitly any index signature.
- forceConsistentCasingInFileNames: Ensures consistent casing in module imports.
- traceResolution: Outputs detailed information about the module resolution process.
- disableSolutionSearching: Disables searching for tsconfig.json files in parent directories when running tsc with a solution file.
- disableReferencedProjectLoad: Prevents loading referenced projects automatically during builds.

ts class - the constructor example (OPP)

TypeScript classes are used regularly in development, especially in scenarios where **object-oriented programming (OOP)** concepts, structure, and type safety are beneficial.

While not always necessary, TypeScript classes are a powerful tool for structuring and scaling applications, especially in OOP-friendly environments or frameworks like Angular or NestJS. However, for functional programming or lightweight solutions, you might rely on alternatives like plain objects and functions.

explanation follow after



ts class - the constructor (OOP)

Defines a class (Course), which includes properties, a constructor, and a method. Here's a detailed explanation:

- The Course **class** defines a blueprint for creating course objects.
- The properties (title, subtitle, creationDate) are explicitly typed, ensuring type safety

Class Definition

typescript Copy Edit

```typescript
class Course {
    title: string;        // A string property to store the course title.
    subtitle: string;     // A string property to store the course subtitle.
    creationDate: Date;   // A Date property to store the creation date of the course.
```

- The **constructor** is a special method automatically called when a new object is instantiated using the class.
- It takes three arguments (title, subtitle, and creationDate) and assigns them to the instance properties of the same name using this.

Constructor

typescript Copy Edit

```typescript
constructor(
    title: string,
    subtitle: string,
    creationDate: Date
) {
    this.title = title;
    this.subtitle = subtitle;
    this.creationDate = creationDate;
}
```

- This **method** calculates the "age" of the course in days.

Method: age

typescript Copy Edit

```typescript
age() {
    const ageInMS =  new Date().getTime() - this.creationDate.getTime();
    return ageInMS / 1000 / 60 / 24;
}
```

- A new **instance** of the Course class is created with the title "TS", subtitle "Bootcamp", and creation date new Date(2001,1,22).

Instantiation

typescript Copy Edit

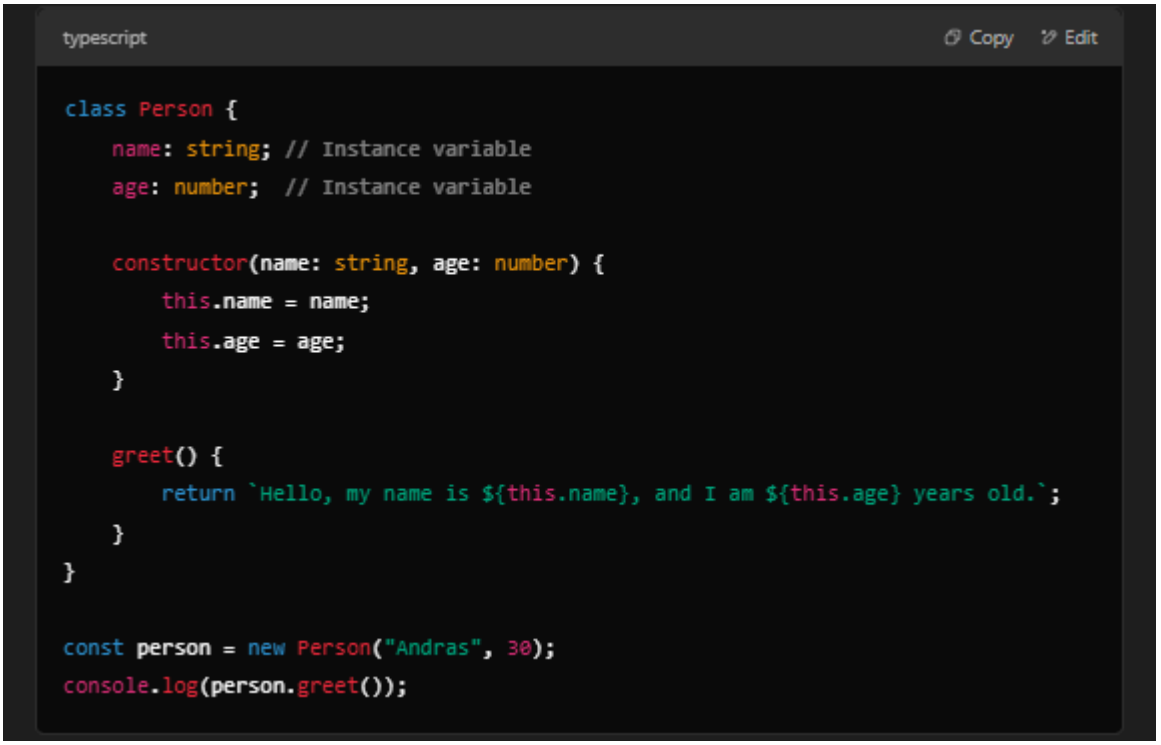
```typescript
const course = new Course("TS", "Bootcamp", new Date(2001,1,22));
```

member variables (or class properties)

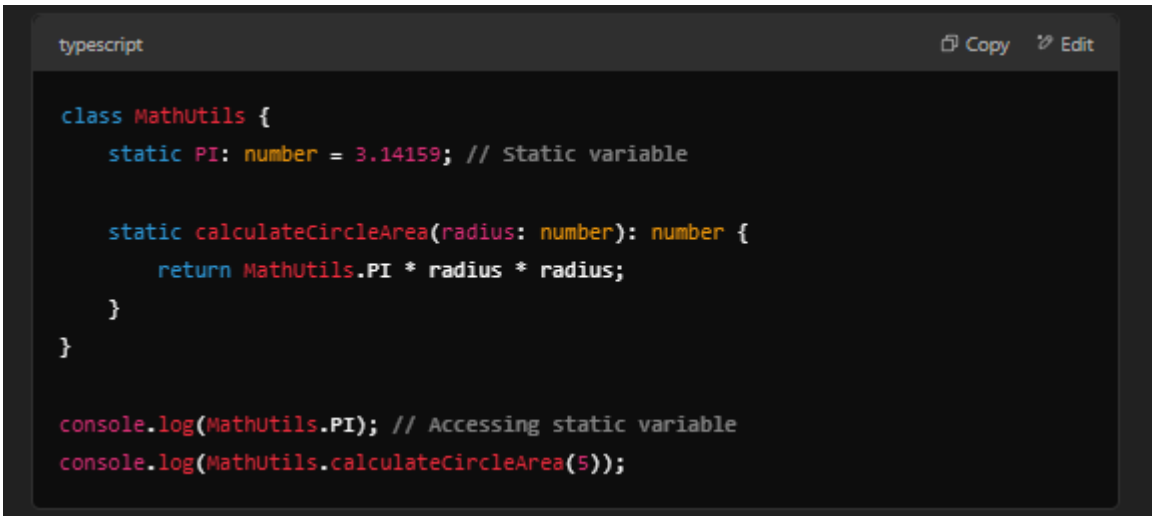
Member variables (or class properties) are variables associated with a class. These member variables represent the state or data that objects created from the class can hold.

Member variables in TypeScript classes define the state of a class. They can have different access levels (public, private, protected) and types (instance, static, readonly, optional). By combining these features with default initialization and parameter properties, TypeScript provides powerful tools for structuring class data.

1. **Instance Variables**
   - Declared inside the class but outside any methods.
   - Each instance of the class has its own copy of these variables.
   - Accessible via this keyword in the class methods.



2. **Static Variables**
   - Declared using the static keyword.
   - Shared among all instances of the class.
   - Accessed using the class name, not this.

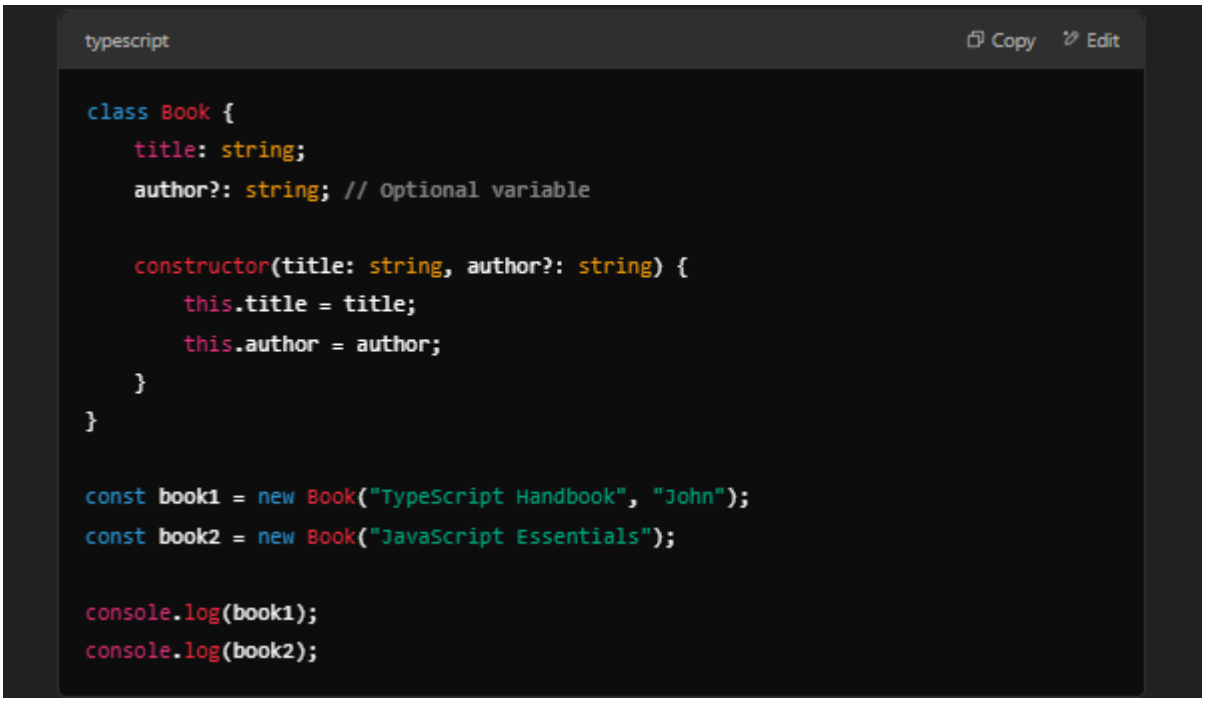


3. Read-Only Variables

- Declared using the static keyword.
- Shared among all instances of the class.
- Accessed using the class name, not this.

4. Optional Variables

- Declared with a ? after the variable name.
- These are optional and may be undefined if not initialize

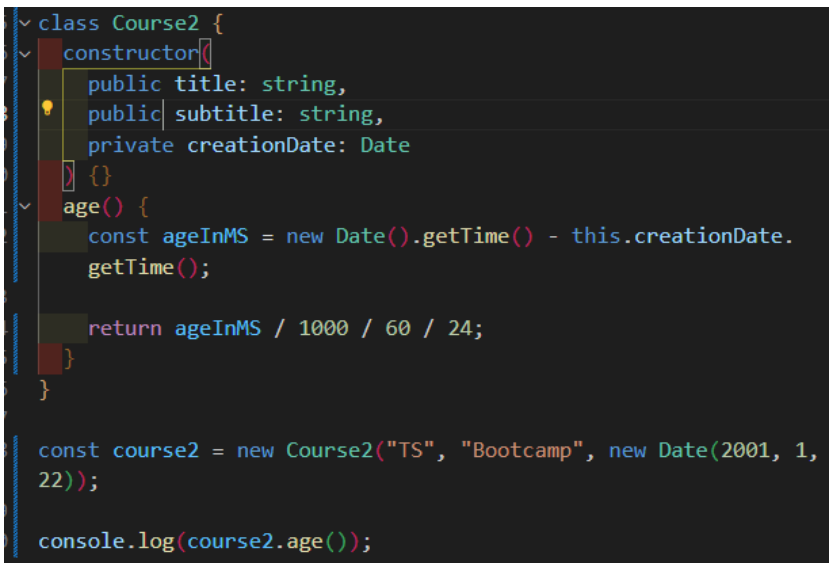


5. Access Modifiers

TypeScript allows control over how member variables are accessed and modified using access modifiers:

- public: Accessible from anywhere (**default behavior**).
- private: Accessible only within the class.
- protected: Accessible within the class and its subclasses.

If we have parameters for each member variable in our constructor with the exact same name as the member variable, we can simplify our syntax. We can remove the declaration of the variable and rely on the constructor to declare our member variables.



getters and setters

In the above example we shouldn't have publicly available properties we should always limit the access.

Getters and setters in TypeScript are special methods that allow you to control access to an object's properties. They provide a way to encapsulate data and define logic for reading (getter) or writing (setter) a property value. These methods are defined within a class and are accessed like properties rather than methods.

getter

```typescript
class Person {
  private _name: string;

  constructor(name: string) {
    this._name = name;
  }

  // Getter method
  get name(): string {
    return this._name;
  }
}

const person = new Person("Andras");
console.log(person.name); // Output: Andras
```

```typescript
//getter
class Course3 {
  constructor(
    private title: string,
    private subtitle: string,
    private creationDate: Date
  ) {}

  get age() {
    const ageInMS = new Date().getTime() - this.creationDate.getTime();

    return ageInMS / 1000 / 60 / 24;
  }
}

const course3 = new Course2("TS", "Bootcamp", new Date(2001, 1, 22));

console.log(course3.age);
```

setter

```typescript
//setter
class Course4 {
  constructor(
    private _title: string,
    private subtitle: string,
    private creationDate: Date
  ) {}

  // Setter for the title
  set title(newTitle: string) {
    if (!newTitle) {
      throw new Error("Title is required");
    } else {
      this._title = newTitle;
    }
  }

  // Getter for the age of the course
  get age() {
    const ageInMS = new Date().getTime() - this.creationDate.getTime();
    return ageInMS / 1000 / 60 / 60 / 24; // Convert ms to days
  }
}

// Usage example
const course4 = new Course4("TS", "Bootcamp", new Date(2001, 1, 22));

// Setting a new title
course4.title = "New value";

// Logging the course details and age
console.log(course4);
console.log(`Course age in days: ${course4.age}`);
```

```
Course4 {
  _title: 'New value',
  subtitle: 'Bootcamp',
  creationDate: 2001-02-22T00:00:00.000Z
}
Course age in days: 8734.742940844908
```

multiple constructors?

TypeScript classes cannot have multiple constructors directly like in some other programming languages (e.g., C# or Java). A TypeScript class can only have one constructor method, and having multiple constructors will result in a compilation error.

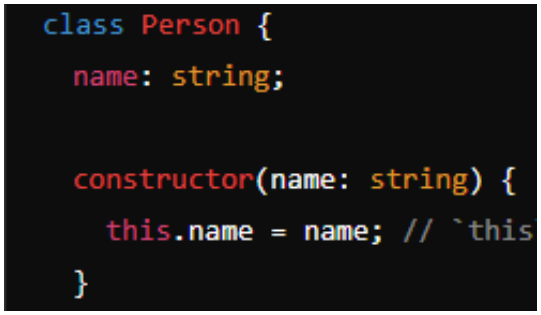
The most common use case when we would need multiple constructors is when we would like to define an initial value for the properties. In this case we could assign it as " " or newDate and the type will be inferred.

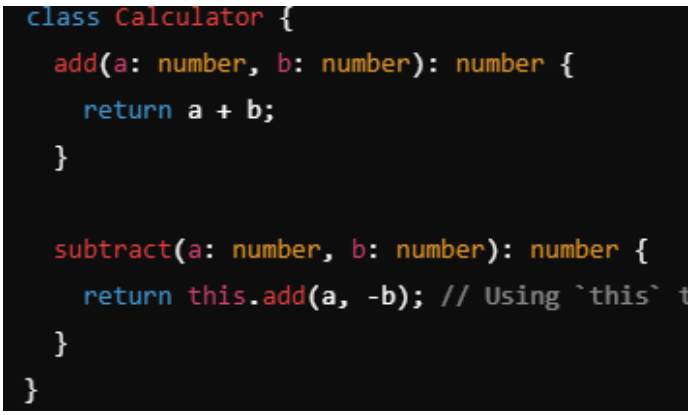
this keyword in class

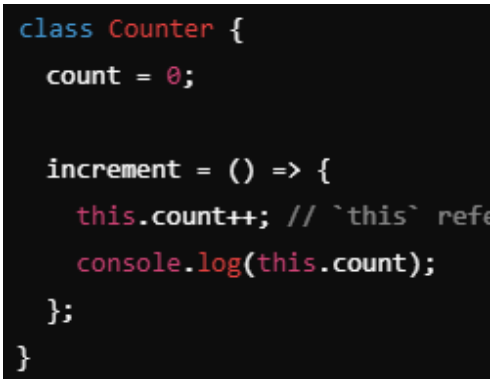
In TypeScript (TS), the **this** keyword is used within a class to refer to the instance of the class that is currently executing the code. It gives access to the properties and methods of that particular instance.

1. **Accessing Properties:** You can use **this** to refer to instance properties and methods declared in the class.
2. **Calling Other Methods: this** can also be used to call other methods of the class.
3. **Arrow Functions and this:** Arrow functions do not have their own this context. Instead, they inherit this from the surrounding context. This is useful when dealing with callbacks or event listeners inside a class.
4. **Context Issues:** If you pass a class method as a callback without binding it, the this context might get lost. You can fix this by explicitly binding the method in the constructor or using an arrow function.
5. **TypeScript-Specific Features:** TypeScript provides strict type checking, which ensures this is used correctly. If this is used outside of the intended context, TypeScript will raise an error.

Examples: 1 -3







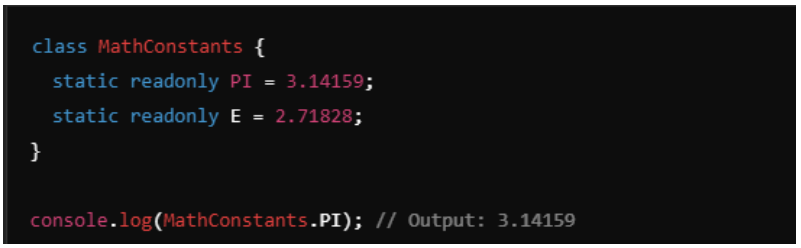
static keyword (properties)

In TypeScript, a static property is a property of a class that is not tied to any specific instance of the class but rather belongs to the class itself. Static properties can be accessed directly on the class without creating an instance of the class.

- **Belongs to the Class:** Static properties are shared across all instances of the class and do not require creating an instance to access them.
- **Access via Class Name:** You access static properties using the class name, not through an instance.
- **Useful for Shared Data:** They are often used for constants, utility methods, or shared data that does not depend on individual object state.

The two most common use case of static properties are:

1. **Defining Constants Specific to a Class:** Static properties are often used to define constants that are tied to the class itself and not any specific instance. These constants are accessible anywhere in the program via the class name.



```
class MathConstants {
  static readonly PI = 3.14159;
  static readonly E = 2.71828;
}

console.log(MathConstants.PI); // Output: 3.14159
```

2. **Implementing Shared, Mutable Variables Across Instances:** Static properties can be used to store data that is shared across all instances of a class. This is useful for keeping track of state or data that is common to all instances. If you want to restrict direct access to these properties (e.g., to make them private), you can combine them with static methods for controlled access.

```
class Counter {
  private static count = 0; // Private static property

  constructor() {
    Counter.count++;
  }

  static getCount(): number {
    return Counter.count; // Public getter for the private property
  }
}

const c1 = new Counter();
const c2 = new Counter();

console.log(Counter.getCount()); // Output: 2
```

static keyword (methods)

A static method in TypeScript is a method that belongs to a class rather than an instance of the class. Like static properties, static methods can be called directly on the class itself without needing to create an instance of the class.

- **Class-Level Scope**: Static methods are invoked on the class itself, not on individual instances.
- **No Access to this:** Since static methods do not operate on class instances, they cannot access instance-specific properties or methods using this. They can only access other static properties or methods of the class.
- **Utility and Shared Behavior:** They are often used for utility functions or operations that are relevant to the class but do not depend on instance-specific data.

```
class MathUtils {
  static PI = 3.14159;

  static calculateCircumference(radius: number): number {
    return 2 * this.PI * radius; // Accessing static property
  }
}

console.log(MathUtils.calculateCircumference(10)); // Output: 62.8318
```

static properties vs methods

Static methods in TypeScript are essentially plain functions that are associated with the **class namespace**. Unlike regular instance methods, they are not tied to an object created from the class.

To call a static method, you must use the **class name** as a prefix. Without this, you won't be able to access the method because it is not part of any specific instance of the class.

Key Differences Between Static and Instance Methods

Aspect	Static Methods	Instance Methods
Belongs to	The class itself	Individual instances of the class
Access	Accessed using the class name	Accessed through an instance of the class
Access to Instance Data	Cannot access instance-specific data or methods	Can access instance properties and methods
Typical Use Case	Utility or shared behavior	Instance-specific operations

inheritance (object oriented)

In object-oriented programming, inheritance is a fundamental concept that allows you to create new classes (called subclasses or derived classes) from existing ones (called superclasses or base classes). This establishes a "parent-child" relationship between the classes, where the subclass inherits properties and methods from the superclass.

extends & super

Base Class (Animal):

- Defines a `name` property and a `makeSound()` method.
- The `constructor` takes the `name` as an argument and assigns it to the `this.name` property.

Derived Class (Dog):

- `extends Animal` indicates that `Dog` inherits from the `Animal` class.
- Defines a `breed` property specific to dogs.
- The `Dog` constructor:
    - Calls `super(name)`: This calls the constructor of the parent class (`Animal`), passing the `name` argument to it. This ensures that the `name` property of the `Animal` class is correctly initialized.
    - Assigns the `breed` value to the `this.breed` property.

Creating an Instance:

- `const myDog = new Dog("Buddy", "Golden Retriever");` creates a new instance of the `Dog` class, passing the name and breed as arguments.

Calling Methods:

- `myDog.makeSound()` calls the `makeSound()` method inherited from the `Animal` class.
- `myDog.bark()` calls the `bark()` method specific to the `Dog` class.

```
class Animal {
  name: string;

  constructor(name: string) {
    this.name = name;
  }

  makeSound(): void {
    console.log("Generic animal sound");
  }
}

class Dog extends Animal {
  breed: string;

  constructor(name: string, breed: string) {
    super(name); // Call the parent class's constructor
    this.breed = breed;
  }

  bark(): void {
    console.log(`${this.name} barks!`);
  }
}

const myDog = new Dog("Buddy", "Golden Retriever");
myDog.makeSound(); // Outputs: "Generic animal sound"
myDog.bark();      // Outputs: "Buddy barks!"
```

protected keyword

- **`private`:** When a class member (property or method) is declared as `private`, it can **only be accessed within the same class**. This enforces strong encapsulation, hiding internal implementation details and preventing accidental or unauthorized modifications from outside the class. Restricting access to the same class
- **`protected`:** Similar to `private`, **`protected`** members are **not accessible from outside the class**. However, they can be accessed by **subclasses** that inherit from the class. This allows for controlled access and flexibility within the class hierarchy. Allows controlled access within the class and its subclasses, promoting code reusability and maintainability within a class hierarchy.

```
class Vehicle {
    protected _wheels: number;

    constructor(wheels: number) {
        this._wheels = wheels;
    }
}

class Car extends Vehicle {
    constructor() {
        super(4); // Access protected _wheels from the parent class
        // ...
    }
}
```

abstract classes (template class)

Abstract classes are a crucial concept in object-oriented programming that provide a blueprint for other classes.

- **Cannot be instantiated directly:** You cannot create instances of an abstract class itself. It's meant to be a base class for other classes to inherit from.
- **May contain abstract methods:** Abstract methods are declared within the abstract class but have no implementation. Subclasses that inherit from the abstract class are **required** to provide concrete implementations for these abstract methods.

```
abstract class Shape {
  abstract getArea(): number;
}

class Circle extends Shape {
  constructor(public radius: number) {
    super();
  }

  getArea(): number {
    return Math.PI * this.radius * this.radius;
  }
}
```

interfaces (object oriented)

Interfaces define a contract or blueprint for classes. They specify the shape of an object by outlining the properties and methods that an object must have.

- **Define contracts:** Interfaces describe the structure of an **object** without providing any implementation details.
- **Used for type checking:** Interfaces are primarily used for **type checking, ensuring** that objects conform to the **specified structure**.
- **Can be implemented by classes:** Classes can implement interfaces, guaranteeing that they have the required properties and methods.

```typescript
// Define an interface
interface Animal {
  name: string;
  makeSound(): void; // A method that the class must implement
}

// Implement the interface in a class
class Dog implements Animal {
  name: string;

  constructor(name: string) {
    this.name = name;
  }

  // Implement the makeSound method
  makeSound(): void {
    console.log(`${this.name} says Woof!`);
  }
}

// Implement another class
class Cat implements Animal {
  name: string;

  constructor(name: string) {
    this.name = name;
  }

  // Implement the makeSound method
  makeSound(): void {
    console.log(`${this.name} says Meow!`);
  }
}

// Instantiate the classes
const dog = new Dog("Buddy");
const cat = new Cat("Whiskers");

dog.makeSound(); // Buddy says Woof!
cat.makeSound(); // Whiskers says Meow!
```

We can also extend interfaces:

```typescript
export interface HasId {
    id:string;
    printId();
}

export interface HasTitle extends HasId {
    title:string;
}
```

singleton (design pattern)

In TypeScript (TS), a singleton is a design pattern that ensures only one instance of a class exists throughout the application's lifecycle.

Key Characteristics:

- **Single Instance:** The core principle is to have a single, globally accessible object of a particular class.
- **Global Access:** This single instance can be accessed from anywhere within the application.
- **Control Over Instantiation:** The singleton class itself controls how and when its single instance is created.

In TypeScript (TS), a singleton is a design pattern that ensures only one instance of a class exists throughout the application's lifecycle.

Explanation:

1. **Private Constructor:** The constructor is made private. This prevents direct instantiation of the class from outside.
2. **Static Instance:** A private static variable `instance` holds the single instance of the class.
3. **getInstance() Method:**
   - This static method acts as the access point to the singleton.
   - It checks if the `instance` exists.
   - If not, it creates a new instance and stores it in the `instance` variable.
   - It always returns the same instance.

```typescript
class Singleton {
  private static instance: Singleton;

  private constructor() {}

  public static getInstance(): Singleton {
    if (!Singleton.instance) {
      Singleton.instance = new Singleton();
    }
    return Singleton.instance;
  }

  // Add your class methods here
  public someMethod(): void {
    console.log("This is the singleton method.");
  }
}
```

```typescript
export class CoursesService {

    private static INSTANCE: CoursesService;

    private constructor() {
        console.log(`The CoursesService was initialized.`);
    }

    static instance() {
        if (!CoursesService.INSTANCE) {
            CoursesService.INSTANCE = new CoursesService();
        }
        return CoursesService.INSTANCE;
    }

}
```

obb finish

There is nothing we cannot do with FB that we can do with OBB however there are some libraries that still use OBB so it's important to know however FB is easier to understand and maintain.

In summary:

- OBB provides a structured way to organize code using classes and objects.
- FP emphasizes pure functions, immutability, and higher-order functions for a more declarative and predictable style.
- TypeScript's flexibility allows you to leverage the strengths of both paradigms to build robust and maintainable applications.

OBB (Object-Oriented Programming) in TypeScript

- **Classes:** TypeScript supports classes, which are blueprints for creating objects. They encapsulate data (properties) and behavior (methods) within a single unit.
- **Inheritance:** Classes can inherit properties and methods from other classes, promoting code reusability and a hierarchical structure.
- **Encapsulation:** Data and methods are often hidden within a class, controlling access and preventing unintended modifications.

FP (Functional Programming) in TypeScript

- **Pure Functions:** Functions that always produce the same output for the same input, without side effects (modifying external state).
- **Immutability:** Data is treated as immutable, meaning it cannot be changed after creation. Changes are handled by creating new data structures.
- **Higher-Order Functions:** Functions that can accept other functions as arguments or return functions as results.

generics

In TypeScript, generics are a powerful feature that allows you to create reusable components that can work with different data types.

Imagine you want to create a function that finds the maximum value in an array. You could write a specific function for an array of numbers:

```
function findMaxNumber(arr: number[]): number {
  if (arr.length === 0) {
    return Number.MIN_VALUE;
  }

  let max = arr[0];
  for (const num of arr) {
    if (num > max) {
      max = num;
    }
  }
  return max;
}
```

But what if you also wanted to find the maximum value in an array of strings (based on alphabetical order)? You'd have to write a separate function:

```
function findMaxString(arr: string[]): string {
  if (arr.length === 0) {
    return "";
  }

  let max = arr[0];
  for (const str of arr) {
    if (str > max) {
      max = str;
    }
  }
  return max;
}
```

This leads to code duplication. Generics provide a solution. In detail later on. :)

Benefits of Generics:

- Code Reusability: Create generic functions and classes that can work with different data types.
- Type Safety: Helps prevent type-related errors at compile time.
- Improved Readability: Makes code more concise and easier to understand.

This is a simplified example, but generics can be used in many ways, such as:

- Creating generic data structures (like generic arrays or linked lists).
- Defining interfaces with type parameters.
- Extending built-in TypeScript types.

commonly uses of generics

Generics are a powerful tool in TypeScript, enabling you to write more flexible, reusable, and type-safe code. Here are some of their common use cases:

1. **Data Structures and Collections**

   a. Arrays: Array<T>: Represents an array of elements of type T.

```
const numbers: Array<number> = [1, 2, 3];
const strings: Array<string> = ["a", "b", "c"];
```

   b. Tuples: [T1, T2, ...Tn]: Represents an array with a fixed number of elements, each with a specific type.

```
const person: [string, number] = ["John Doe", 30]; // [name: string, age: number]
```

   c. Objects: You can define interfaces or types with generic properties.

```
interface KeyValuePair<K, V> {
    key: K;
    value: V;
}

const keyValue: KeyValuePair<string, number> = { key: "name", value: 25 };
```

2. **Utility Functions**

   a. Creating generic helper functions

```
function identity<T>(arg: T): T {
    return arg;
}

const result1 = identity<string>("hello"); // result1 is of type string
const result2 = identity<number>(42); // result2 is of type number
```

   b. Higher-order functions: Functions that take other functions as arguments or return functions.

```
function compose<A, B, C>(f: (b: B) => C, g: (a: A) => B): (a: A) => C {
    return (a: A) => f(g(a));
}
```

3. **Classes and Interfaces**

   a. Generic classes: Define classes that can work with different data types.

```
class Stack<T> {
    private items: T[] = [];

    push(item: T): void {
        this.items.push(item);
    }

    pop(): T | undefined {
        return this.items.pop();
    }
}
```

   b. Generic interfaces: Define interfaces that can be implemented by classes with different data types.

```
interface Comparable<T> {
    compareTo(other: T): number; // -1 if less than, 0 if equal, 1 if greater than
}
```

4. **Extending Built-in Types**
   (Creating custom array-like types)

```
type ReadonlyArray<T> = Array<Readonly<T>>; // Array of readonly elements
```

partial interface (optional)

An optional interface is a way to define an interface where some properties are not required. This flexibility allows you to create more adaptable and reusable interfaces.

Partial is a utility type in TypeScript that makes all properties of the given interface optional. In this case, Partial<Course> creates an interface where title, subtitle, and lessonsCount are all optional. This allows you to pass only the properties you want to update to the updateCourse function.

Benefits of using `Partial`:

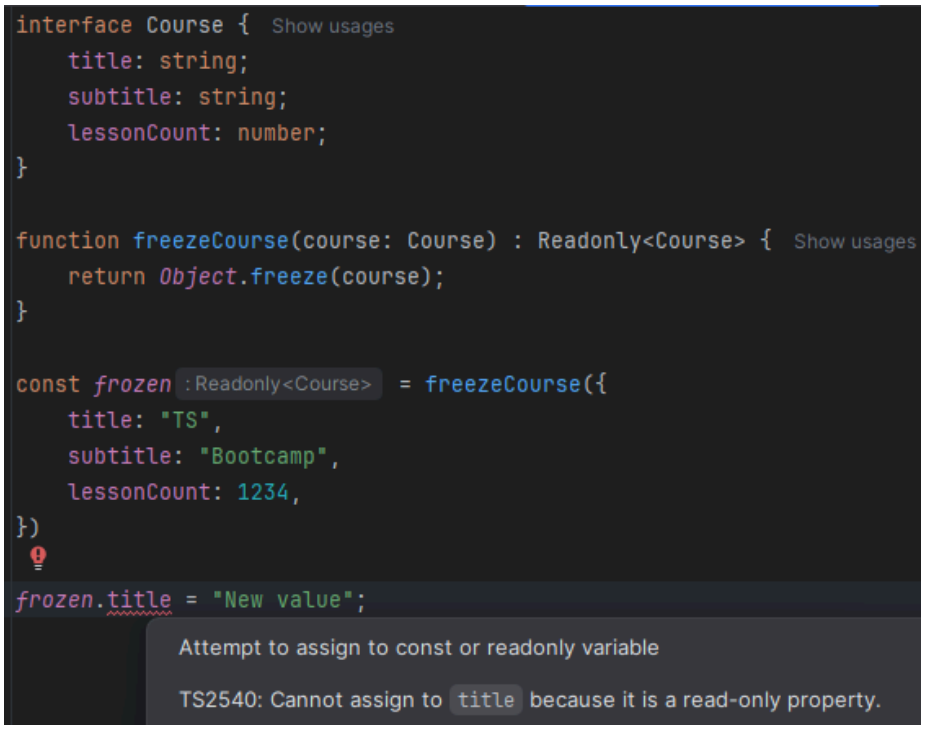
- **Flexibility:** Allows you to update only the necessary properties of a course without needing to provide values for all properties.
- **Type Safety:** Ensures that the provided update object only contains properties that are valid for the `Course` interface.
- **Readability:** Improves code clarity by explicitly indicating which properties are being updated.

```
export interface  Course {  Show usages
    title: string;
    subtitle: string;
    lessonsCount: number;
}

export function updateCourse(courseId: string, update: Partial<Course>) :void  {

}

updateCourse("1", {
    title: "New title",
    }
)

updateCourse("1", {
        subtitle: "New subtitle",
    }
)
```

read only interface

In TypeScript, a read-only interface is a special type that makes all properties of an interface immutable. This means that once an object conforms to a read-only interface, its properties cannot be modified.

- **`Readonly<T>`:** This is a utility type provided by TypeScript. It takes any type `T` as input and creates a new type where all properties of `T` are marked as read-only.
- **`Readonly<Course>`:** In this specific case, `Readonly<Course>` creates a new type based on the `Course` interface. However, all properties (`title`, `subtitle`, `lessonCount`) in this new type become immutable.
- **`freezeCourse` function:** This function returns an object of type `Readonly<Course>`. This means that any object returned by this function cannot have its properties modified.



generic functions

Imagine in the above example if we have multiple interfaces and we want them to be readonly as well. We would need to write nearly duplicate code.

```
export function freezeCourse(course:Course): Readonly<Course> {
    return Object.freeze(course);
}

function freezeLesson(lesson:Lesson): Readonly<Lesson> {
    return Object.freeze(lesson);
}

const frozenCourse = freezeCourse({
```

A **generic function** is a function that can work with **multiple types** instead of being restricted to a single type. It provides flexibility while ensuring type safety.

The function freeze**<T> is a generic function**. The <T> syntax denotes that this function accepts a type parameter T, meaning it can be used with any type.

- `<T>`: Declares a generic type `T` that can be replaced with any type.
- `input: T`: The function takes an argument of type `T`.
- `Readonly<T>`: The return type is `Readonly<T>`, meaning that the properties of `T` become **immutable** (cannot be modified).
- `Object.freeze(input)`: Freezes the input object, preventing modifications.

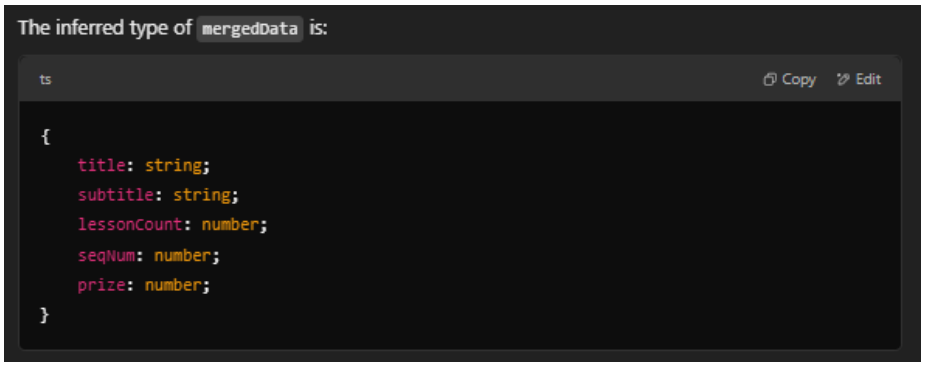
```
interface Course {  Show usages
    title: string;
    subtitle: string;
    lessonCount: number;
}

interface Lesson {  Show usages
    title: string;
    seqNum: number;
}

function freeze<T>(input: T): Readonly<T>{  Show usages
return Object.freeze(input)
}

const frozenCourse :Readonly<Course>  = freeze<Course> ({
    title: "new title",
    subtitle: "new subtitle",
    lessonCount: 321,
})

const frozenLesson :Readonly<Lesson>  = freeze<Lesson> ({
    title: "new title",
    seqNum: 43
})
```

generic functions with generic parameters (merge)

This example although would be a merge, but wouldn't be type safe:

A generic function allows us to create reusable, type-safe logic that can work with multiple types. In this case, we want to merge two objects while maintaining their type information.

1. **Generic Type Parameters `<T, U>`**:
   - `T` represents the type of the first object (`obj1`).
   - `U` represents the type of the second object (`obj2`).
   - This allows the function to accept two objects of different types.
2. **`Object.assign(obj1, obj2)`**:
   - This merges `obj2` into `obj1`, meaning all properties from `obj2` get added to `obj1`.
3. **`as T & U`**:
   - The return type is a **TypeScript intersection type** (`T & U`), which means it combines both types into one.



generic type constraints

It would nothing prevent us from the above example to for example freeze a string or boolean or any other primitive type:

```
function freeze<T>(input: T): Readonly<T>{  Show usages  Andras E
return Object.freeze(input)
}

const frozenCourse :Readonly<Course>  = freeze<Course> ({
    title: "new title",
    subtitle: "new subtitle",
    lessonCount: 321,
})

const frozenLesson :Readonly<Lesson>  = freeze<Lesson> ({
    title: "new title",
    seqNum: 43
})

freeze(true)
```

Generic type constraints in TypeScript can be used to enforce specific requirements on the types that can be used with a generic function. This helps to improve type safety and prevent unexpected errors.

The extends object constraint ensures that the freeze function can only be used with objects.

```
function freeze<T extends object>(input: T): Readonly<T>{
return Object.freeze(input)
}

const frozenCourse :Readonly<Course>  = freeze<Course> ({
    title: "new title",
    subtitle: "new subtitle",
    lessonCount: 321,
})

const frozenLesson :Readonly<Lesson>  = freeze<Lesson> ({
    title: "new title",
    seqNum: 43
})

freeze(true)
```

1. **Interface Definition:**
   - We define an interface `Course` to represent the structure of our course object.

2. **Generic Freeze Function:**
   - `freeze<T extends object>(input: T): Readonly<T>`:
   - `T extends object`: This generic type constraint ensures that the function can only be used with objects.
   - `input: T`: The function accepts an object of type `T`.
   - `Readonly<T>`: The function returns a read-only version of the input object. This means that any attempt to modify the properties of the returned object will result in a compile-time error.

3. **Usage:**
   - `const frozenCourse = freeze<Course>({ ... });`:
     - We call the `freeze` function with a `Course` object and specify the type parameter as `Course`.
     - The `freeze` function returns a read-only `Course` object.

4. **Attempting to Modify:**
   - `frozenCourse.title = "updated title";`: This line will result in a compile-time error because the `frozenCourse` object is read-only.

key of constraints

keyof T: This is a utility type in TypeScript. Given a type T, keyof T returns a union of all the possible property names of that type. This example demonstrates how the keyof constraint can be used to create a generic function that safely accesses properties of an object, preventing potential runtime errors.

1. **Interface Definition:**
   - We define an interface `User` with two properties: `name` (string) and `age` (number).
2. **`keyof` Constraint:**
   - `type UserKeys = keyof User;`: This line defines a type alias `UserKeys` that represents the union of all property names of the `User` interface. In this case, `UserKeys` is `"name" | "age"`.
3. **Generic Function:**
   - `function getProperty<T, K extends keyof T>(obj: T, key: K): T[K]`:
   - `T`: Generic type representing the object type.
   - `K extends keyof T`: This is the key constraint. It ensures that the `key` argument must be one of the valid property names of the object `T`.
   - `T[K]`: This is a lookup type. It returns the type of the property specified by the key `K` in the object `T`.

Usage:

- `const userName = getProperty(user, "name");`: This line extracts the value of the `name` property from the `user` object. TypeScript infers that `userName` will be of type `string`.
- `const userAge = getProperty(user, "age");`: Similarly, this line extracts the `age` property, and `userAge` will be of type `number`.
- The commented-out line `getProperty(user, "address")` would result in a compile-time error because `address` is not a valid property of the `User` interface.

```
interface User {
  name: string;
  age: number;
}

type UserKeys = keyof User; // UserKeys is now "name" | "age"

function getProperty<T, K extends keyof T>(obj: T, key: K): T[K] {
  return obj[key];
}

const user: User = { name: "Alice", age: 30 };

const userName = getProperty(user, "name"); // userName is of type string
const userAge = getProperty(user, "age"); // userAge is of type number

// This will not compile:
// const invalid = getProperty(user, "address"); // Property 'address' does not ex
```

generic classes

In TypeScript, generic classes allow you to create classes that can work with a wide range of data types. These type parameters can then be used within the class to define the types of properties, methods, and other members.

1. **class KeyValue<K, V>**
   - This declares a class named KeyValue.
   - <K, V> are the generic type parameters. K represents the type of the key, and V represents the type of the value.

2. **constructor(public readonly key: K, public readonly value: V) {}**
   - This defines the constructor of the KeyValue class.
   - public readonly: This makes the key and value properties publicly accessible but read-only (they cannot be modified after the object is created).
   - The constructor takes two arguments: key of type K and value of type V

3. **print() { ... }**
   - This is a method within the class that logs the key and value to the console.

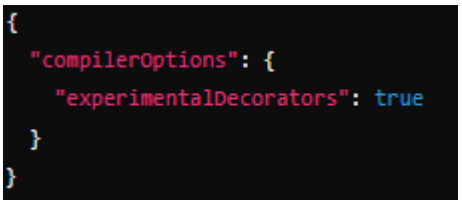
4. **Usage Examples**
   - const p1 = new KeyValue("1", 10)
     - Creates an instance of KeyValue where K is string and V is number.
     - p1.key will be of type string and p1.value will be of type number.
   - const p2 = new KeyValue("2", "TS")
     - Creates another instance of KeyValue where K is string and V is string.
     - p2.key and p2.value will both be of type string.

```
class KeyValue<K, V> {  Show usages

    constructor(  Show usages
        public readonly key: K,
        public readonly value: V) {}

        print() :void  {  no usages
            console.log(`key value: ${this.key} value: ${this.value}`);
        }
}

const p1 = new KeyValue("1", 10)

const val1 :number  = p1.value

const p2 = new KeyValue("2", "TS")

const val2 :string  = p2.value
```

decorators (optional, not widely used)

In TypeScript, decorators are a special kind of declaration that can be attached to classes, methods, properties, accessors, or parameters to modify their behavior. Decorators provide a way to apply reusable logic and. Decorators are powerful tools that allow us to modify behavior at runtime. They are particularly useful in frameworks like Angular for dependency injection, metadata storage, and validation.

Decorators are **functions** that are prefixed with the `@` symbol and applied to a class or its members. **Note:** To use decorators, you must enable the `experimentalDecorators` flag in `tsconfig.json`:



Plain TypeScript object inheritance, even though it's a convenient way of adding existing functionality to one of our classes. This is not a very flexible mechanism because if we have multiple libraries, each one providing its own functionality and we would like to enrich our DB service class with the features of those multiple libraries, there is really no way of doing that with plain TypeScript object inheritance. What we can do instead is to use decorators and something known as method programming.

For example, we can extend a class with **BaseDatabasService** but if we would like to use multiple libraries, for example include a logging in or performance monitoring, we won't be able to without decorators.

```
class DbService extends BaseDatabasService{

   saveData(data: any) {

        console.log(`saving data $data`);

   }

}
```

As you can see, decorators are a very powerful feature of the typescript language.They allow us to add existing functionality into a new class by weaving that functionality together without having to use inheritance. And like when using inheritance with decorators, we can easily compose and weave together functionality from many different libraries and frameworks together in one single class or method.

class decorator

A class decorator is a function applied to a class that can modify or extend its behavior. Here, `@Logger` is applied to the `Person` class. When the class is **declared**, the `Logger` function is executed.

```
function Logger(constructor: Function) {
  console.log(`Class created: ${constructor.name}`);
}

@Logger
class Person {
  constructor(public name: string) {}
}

const p1 = new Person("Andras");
// Output: "Class created: Person"
```

method decorator

A method decorator modifies a method's behavior. Here, `@Log` wraps the `add` method, logging its execution.

```
function Log(target: any, methodName: string, descriptor: PropertyDescriptor) {
  const originalMethod = descriptor.value;

  descriptor.value = function (...args: any[]) {
    console.log(`Calling ${methodName} with arguments:`, args);
    return originalMethod.apply(this, args);
  };

  return descriptor;
}

class Calculator {
  @Log
  add(a: number, b: number) {
    return a + b;
  }
}

const calc = new Calculator();
calc.add(2, 3);
// Output:
// Calling add with arguments: [2, 3]
```

property decorator

A property decorator can be used to **track changes** or **add metadata** to properties. The decorator simply logs that the `name` property is required.

```
function Required(target: any, propertyName: string) {
  console.log(`Property "${propertyName}" is marked as required.`);
}

class User {
  @Required
  name: string;

  constructor(name: string) {
    this.name = name;
  }
}

// Output: "Property 'name' is marked as required."
```

accessor decorator

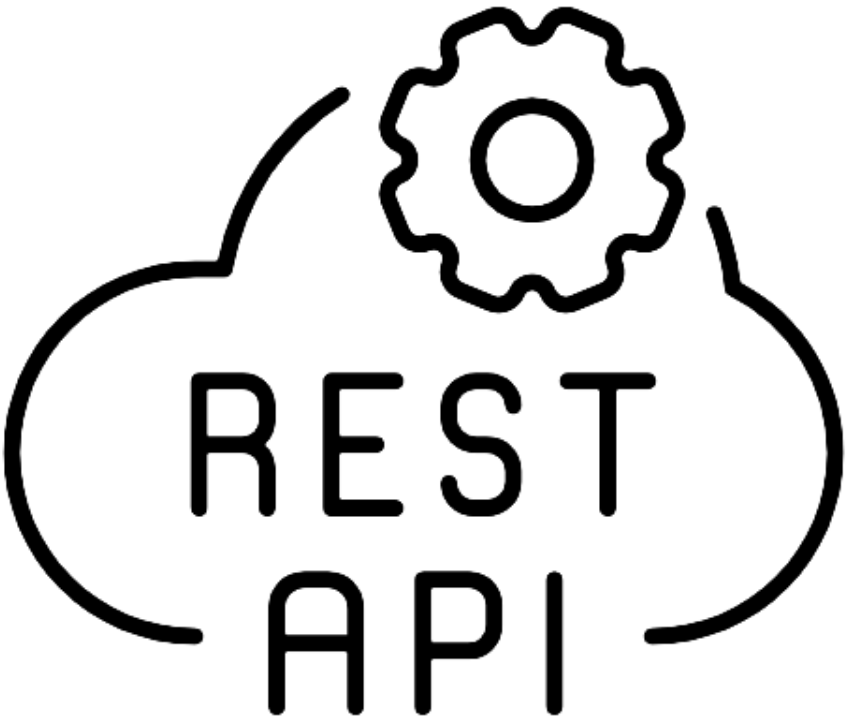
An accessor decorator is used on **getters and setters**.
The decorator ensures that the `balance` value is never negative.

```ts
function Validate(target: any, methodName: string, descriptor: PropertyDescriptor) {
  const setter = descriptor.set;

  descriptor.set = function (value: number) {
    if (value < 0) {
      throw new Error("Value cannot be negative");
    }
    setter!.call(this, value);
  };
}

class Account {
  private _balance: number = 0;

  @Validate
  set balance(amount: number) {
    this._balance = amount;
  }

  get balance() {
    return this._balance;
  }
}

const account = new Account();
account.balance = 100; // Works fine
// account.balance = -50; // Throws an error: "Value cannot be negative"
```

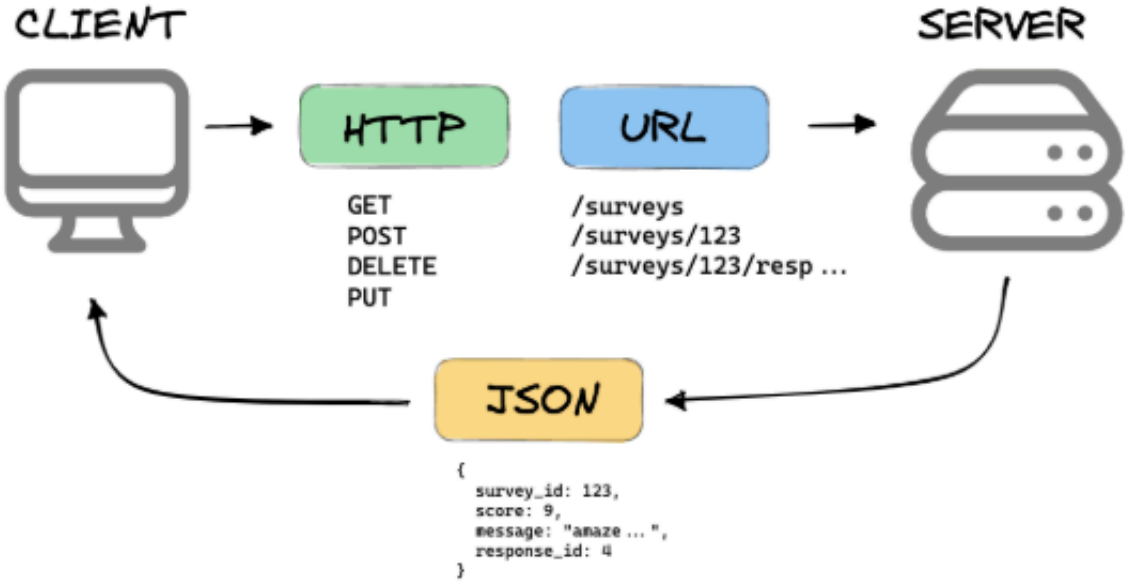
parameter decorator

A parameter decorator can **track function parameters**.
Here, `@LogParam` tracks the parameter's index in the `greet` method.

```ts
function LogParam(target: any, methodName: string, paramIndex: number) {
  console.log(`Parameter at index ${paramIndex} in ${methodName} is logged.`);
}

class Person {
  greet(@LogParam message: string) {
    console.log(message);
  }
}

// Output: "Parameter at index 0 in greet is logged."
```



WHAT IS A REST API?



rest-api project setup

REST API built with **TypeScript** is a small backend service that allows users (clients) to **send and receive data** over the internet using HTTP requests. It typically runs on a server (e.g., **Node.js with Express**) and interacts with a database or in-memory data.

- **REST API (Representational State Transfer)** is a way for clients (browsers, mobile apps, other servers) to interact with a backend.
- It follows the **HTTP protocol** (using methods like `GET`, `POST`, `PUT`, `DELETE`).
- **TypeScript** adds strong typing and better error handling to make development safer.

1. $ npm init~
2. $ npm i typescript
3. tsconfig.json

```
{
  "compilerOptions": {
    "target": "ES6",
    "outDir": "dist"
  }
}
```

4. new folder src and new file server.ts
5. package.json

```
{
  "name": "rest-api",
  "version": "1.0.0",
  "description": "",
  "scripts": {
    "build": "tsc",
    "start-server": "node dist/server.js"
  },
  "author": "",
  "license": "ISC",
  "dependencies": {
    "typescript": "^5.7.3"
  }
}
```

6. $ tsc
7. $ npm run start-server (testing hello world log from server.ts)

```
> rest-api@1.0.0 start-server
> node dist/server.js

hellooo world
```

8. $ npm i express
9. $ npm i --save-dev @types/express (as express doesn't have built in ts support so need to install @types)
10. new folder src/routes (Each endpoint, also known in Express terminology as a route should be written in its own separate file)

11. Create src/routes folder and a root.ts

12. setup express

```ts
import express from "express";
import {root} from "./routes/root";

const app :Express  = express();

function setupExpress() :void {
    // http://localhost:3000/
    app.route("/").get(root);
}

function startServer() :void {
    app.listen(3000, () :void => {
        console.log("Server started on port 3000");
    });
}

setupExpress();

startServer();
```

13. setup root.ts

```ts
import {Request, Response} from "express";

export function root(request: Request, response: Response) :void {
    response.status(200).send("<h1>Hello World Express!</h1>");
}
```

14. $ npm run build

```
> rest-api@1.0.0 build
> tsc
```

15. $ node dist/server.js

```
Server started on port 3000
```

Hello World Express!

PS.: I updated tsconfig.json to ESNext and added

```json
{
  "compilerOptions": {
    "target": "ESNext",
    "outDir": "dist",
    "esModuleInterop": true,
    "moduleResolution": "nodenext",
    "module": "NodeNext"
  }
}
```

- `"target": "ESNext"` → Uses the latest JavaScript features.
- `"module": "NodeNext"` → Enables correct handling of ES modules in Node.js.
- `"moduleResolution": "NodeNext"` → Ensures TypeScript resolves modules correctly.
- `"esModuleInterop": true` → Allows `import express from "express";` instead of `import * as express from "express";` (didn`t work w ES6 or ESNext only ES5)

clean start & hot reload // configuring port

16. $ npm i --save-dev rimraf & add to package.json (slang for rm -rf)
a simple package that cleans dist folder before each build process

```
"scripts": {
  "clean": "rimraf dist",
```

17. $ npm install npm-run-all --save-dev // package that allow us to run packages in sequence or parallel, so we will clean dist folder, build project and run server (on windows on mac there we could do all that with &&)

```
"scripts": {
  "clean": "rimraf dist",
  "build": "tsc",
  "start-server": "node dist/server.js",
  "dev": "npm-run-all clean build start-server"
```

18. $ npm install --save-dev tsc-watch // will allow us hot-reloading

```
"scripts": {
  "clean": "rimraf dist",
  "build": "tsc",
  "start-server": "node dist/server.js",
  "start-dev-server": "tsc-watch --onSuccess \"node dist/server.js\"",
  "dev": "npm-run-all clean build start-dev-server"
```

19. $ npm run dev

20. edit the possibility of configuring the port number where our server is running using a command line argument.
Creating utils.ts next to server.ts

```
export function isInteger(input: string) {
  return input?.match(/^\d+$/) ?? false;
}
```

Edit server.ts

```
function startServer() {
  let port: number;

  const portArg = process.argv[2];

  if (isInteger(portArg)) {
    port = parseInt(portArg);
  }

  if (!port) {
    port = 3000;
  }

  app.listen(port, () => {
    console.log(`Server started on port ${port}`);
  });
}
```

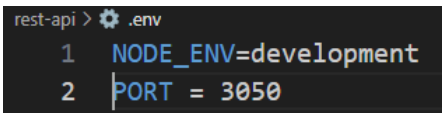
server can run now on any port number, if it is not defined or not a proper integer it will be running on default 3000 port, for example i run it here instead of default 3000 on 5050:

```
"start-dev-server": "tsc-watch --onSuccess \"node dist/server.js 5050\"",
```

```
19:27:51 - Found 0 errors. Watching for file changes.
Server started on port 5050
```

This is going to make it very convenient for your server to run in different environments such as development, testing, staging or production.

21. $ npm i dotenv // setting up multiple environments for test/deploy etc. Adding dotenv files to root of the project and as soon as we import it in server.ts




```
import * as dotenv from "dotenv";

const results = dotenv.config();

if (results.error) {
  console.log("Error loading .env file");
  process.exit(1);
  ``;
}
```

22. Going to support the two ways of receiving the port number. First, we're going to give priority to an environment variable. So if we set it via an environment variable, that's the one that is going to be used to start up the server. If there is no environment variable available, then we are going to check a command line argument. And finally, if neither of those are present, then we're going to go for a default port number.

```
function startServer() {
  let port: number;

  const portEnv = process.env.PORT;
  const portArg = process.argv[2];

  if (isInteger(portEnv)) {
    port = parseInt(portEnv);
  }

  if (!port && isInteger(portArg)) {
    port = parseInt(portArg);
  }

  if (!port) {
    port = 3000;
  }

  app.listen(port, () => {
    console.log(`Server started on port ${port}`);
  });
}
```

23. Adding winston logging library. Typically we only want to have console.log statements in our development environment but in production we only use it for error logging.
$ npm i winston // adding it to dotenv, we could have multiple levels, debug, error, info etc.

```
NODE_ENV=development
#PORT = 3000
LOGGER_LEVEL=debug
```

24. Creating a logger.ts file in src folder where we set up winston

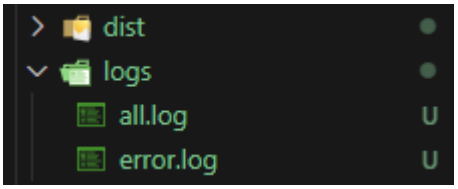
```ts
import * as winston from "winston";

export const logger = winston.createLogger({
  level: process.env.LOGGER_LEVEL,
  format: winston.format.json({
    space: 4,
  }),
  transports: [
    new winston.transports.File({
      filename: "logs/all.log",
    }),
    new winston.transports.File({
      filename: "logs/error.log",
      level: "error",
    }),
  ],
});
```

```ts
if (process.env.NODE_ENV !== "production") {
  logger.add(
    new winston.transports.Console({
      format: winston.format.simple(),
    })
  );
}
```

25. Replacing console.log statements with Winston's logger. Please keep in mind that the import order is crucial—Winston must be imported after dotenv. Also, note that the first console.log should remain, as Winston has not been initialized at that point.

```ts
import * as dotenv from "dotenv";

const results = dotenv.config();

if (results.error) {
  console.log("Error loading .env file");
  process.exit(1);
  ``;
}

import express from "express";
import { root } from "./routes/root";
import { isInteger } from "./utils";
import { logger } from "./logger";
```

```
10:47:06 - Starting compilation in watch mode...

10:47:13 - Found 0 errors. Watching for file changes.
info: Server started on port 3000
```



26. Setup Render & new postgresql database https://dashboard.render.com/

27. Install https://www.postgresql.org/download/windows/
You may need to do more setup to show up in vs-code terminal

If PostgreSQL is installed but psql isn't recognized, follow these steps:

- Option 1: Restart Your Terminal & Try Again
  Sometimes, a simple restart of VS Code or your PC is enough.

- Option 2: Manually Add PostgreSQL to PATH
  - Find PostgreSQL Installation Path
    Open File Explorer
    Navigate to:
    C:\Program Files\PostgreSQL\{your_version}\bin
    Example:
    C:\Program Files\PostgreSQL\15\bin
    Copy this path

  - Add It to the System PATH
    Press Win + R, type sysdm.cpl, and press Enter
    Go to Advanced → Click Environment Variables
    Under System Variables, find Path → Click Edit
    Click New, then paste the copied PostgreSQL bin path
    Click OK to save
    Restart VS Code / PowerShell
    Open a new terminal and try:
    psql --version

28. VS-code terminal psql External Database URL for example:
psql postgresql://rest_api_7slp_user:E9tJa7qMUm@dpg3dm0-a.frankfurt-postgres.render.com/rest_api_7slp

29. Once connected $ /dt;

```
rest_api_7slp=> \dt;
Did not find any relations.
```

30. Adding database library - TypeORM (Object relational mapping)
$ npm i typeorm

31. Installing database driver for the Postgres database
$ npm i pg

32. Now that we have all our dependencies installed, we need to set up our environment to support different databases. For example, in development, we might be connecting to a development database that only one developer has access to while in production. We are connected, of course, to the production database that is live and to which many users are connecting. So in order to do that, we're going to add here to our environment variable file a couple of new variables. Adding to our environment variable file a couple of new variables. Should look something like this (Render used)

.env

```
NODE_ENV=development
#PORT = 3000
LOGGER_LEVEL=debug

DB_HOST=dpg-cujp3uij1k6c73d0g2m0-a
DB_PORT=5432
DB_USERNAME=rest_api_7slp_user
DB_PASSWORD=Eevy4yI8J4lQ6Q8cfpFF9tJQSah7qMUm
DB_NAME=rest_api_7slp
```

33. Creating new file in /src folder called data-source.ts fill out with env variables

src > TS data-source.ts > ...
C:\Users\rohad\WebstormProjects\TS-Bootcamp\rest-api\src • Contains emphasized items

```
export const dataSource = new DataSource({
  type: "postgres",
  host: process.env.DB_HOST,
  username: process.env.DB_USERNAME,
  password: process.env.DB_PASSWORD,
  port: parseInt(process.env.DB_PORT),
  database: process.env.DB_NAME,
  ssl: true,
  entities: [],
  logging: true,
});
```

34. Edit server.ts moving setupExpress and startServer // if datasource is initialized we start the server

```
AppDataSource.initialize()
  .then(() => {
    logger.info("Data source initialized");
    setupExpress();
    startServer();
  })
  .catch((error) => {
    logger.error("Error initializing data source", error);
    process.exit(1);
  });
```

35. Create a new folder called model in src and new files that will contain all our database model classes.
And turn it into TypeORM model (Entity is a decorator from typeorm, we need to add the flag to allow decoratorns on tsconfig.json

```
"experimentalDecorators": true
```

```
export class Course {
  id: number;
  seqNo: number;
  title: string;
  iconUrl: string;
  longDescription: string;
  category: string;
}
```

```
import { Entity } from "typeorm";

@Entity()
export class Course {
  id: number;
  seqNo: number;
  title: string;
  iconUrl: string;
  longDescription: string;
  category: string;
}
```

36. We add a couple of more decorators and setup typeORM, adding new model lessons.ts

```
import {
  Column,
  CreateDateColumn,
  Entity,
  PrimaryGeneratedColumn,
  UpdateDateColumn,
} from "typeorm";

@Entity({
  name: "COURSES",
})
export class Course {
  @PrimaryGeneratedColumn()
  id: number;

  @Column()
  seqNo: number;

  @Column()
  title: string;

  @Column()
  longDescription: string;

  @Column()
  category: string;

  @CreateDateColumn()
  createdDate: Date;

  @UpdateDateColumn()
  lastModifiedDate: Date;
}
```

```
import {
  Column,
  CreateDateColumn,
  Entity,
  PrimaryColumn,
  UpdateDateColumn,
} from "typeorm";

@Entity({
  name: "LESSONS",
})
export class Lesson {
  @PrimaryColumn()
  id: number;

  @Column()
  title: string;

  @Column()
  duration: string;

  @Column()
  seqNo: number;

  @CreateDateColumn()
  createdDate: Date;

  @UpdateDateColumn()
  lastModifiedDate: Date;
}
```

37. We are now going to link them together in a one to many, many to one bidirectional relationship between the two models. One to many and many to one.

```ts
import {
  Column,
  CreateDateColumn,
  Entity,
  OneToMany,
  PrimaryGeneratedColumn,
  UpdateDateColumn,
} from "typeorm";
import { Lesson } from "./lesson";
@Entity({
  name: "COURSES",
})
export class Course {
  @PrimaryGeneratedColumn()
  id: number;

  @Column()
  seqNo: number;

  @Column()
  title: string;

  @Column()
  longDescription: string;

  @Column()
  category: string;

  @OneToMany(() => Lesson, (lesson) => lesson.course)
  lessons: Lesson[];

  @CreateDateColumn()
  createdDate: Date;

  @UpdateDateColumn()
  lastModifiedDate: Date;
}
```

```ts
import {
  Column,
  CreateDateColumn,
  Entity,
  JoinColumn,
  ManyToOne,
  PrimaryColumn,
  UpdateDateColumn,
} from "typeorm";
import { Course } from "./course";

@Entity({
  name: "LESSONS",
})
export class Lesson {
  @PrimaryColumn()
  id: number;

  @Column()
  title: string;

  @Column()
  duration: string;

  @Column()
  seqNo: number;

  @ManyToOne(() => Course, (course) => course.lessons)
  @JoinColumn({ name: "courseId" })
  course: Course;

  @CreateDateColumn()
  createdDate: Date;

  @UpdateDateColumn()
  lastModifiedDate: Date;
}
```

and modifying the data-source.ts adding entities and set synchronise true (just for development)

```ts
import { DataSource } from "typeorm";
import { Course } from "./models/course";
import { Lesson } from "./models/lesson";

export const AppDataSource = new DataSource({
  type: "postgres",
  host: process.env.DB_HOST,
  username: process.env.DB_USERNAME,
  password: process.env.DB_PASSWORD,
  port: parseInt(process.env.DB_PORT),
  database: process.env.DB_NAME,
  ssl: true,
  entities: [Course, Lesson],
  synchronize: true,
  logging: true,
});
```

38. npm i reflect-metadata (for typeORM Decorators and Decorator Metadata) and import it in server.ts

```ts
import "reflect-metadata";
import * as express from 'express';
import {root} from "./routes/root";
import {isInteger} from "./utils";
import {logger} from "./logger";
import {AppDataSource} from "./data-source";
```

39. Testing setup
npm run dev

```
12:27:18 - Found 0 errors. Watching for file changes.
query: SELECT * FROM current_schema()
query: SELECT version();
query: START TRANSACTION
query: SELECT * FROM current_schema()
query: SELECT * FROM current_database()
query: SELECT "table_schema", "table_name", obj_description(('"' || "table_schema" || '"."' || "table_name" || '"')::regclass, 'pg_class') AS tabl
e_comment FROM "information_schema"."tables" WHERE ("table_schema" = 'public' AND "table_name" = 'LESSONS') OR ("table_schema" = 'public' AND "tab
le_name" = 'COURSES')
query: SELECT * FROM "information_schema"."tables" WHERE "table_schema" = 'public' AND "table_name" = 'typeorm_metadata'
query: CREATE TABLE "LESSONS" ("id" integer NOT NULL, "title" character varying NOT NULL, "duration" character varying NOT NULL, "seqNo" integer N
OT NULL, "createdDate" TIMESTAMP NOT NULL DEFAULT now(), "lastModifiedDate" TIMESTAMP NOT NULL DEFAULT now(), "courseId" integer, CONSTRAINT "PK_8
72e7d4036724c35c8804100943" PRIMARY KEY ("id"))
query: CREATE TABLE "COURSES" ("id" SERIAL NOT NULL, "seqNo" integer NOT NULL, "title" character varying NOT NULL, "longDescription" character var
ying NOT NULL, "category" character varying NOT NULL, "createdDate" TIMESTAMP NOT NULL DEFAULT now(), "lastModifiedDate" TIMESTAMP NOT NULL DEFAUL
T now(), CONSTRAINT "PK_27fddb82290e2c8378be8159ef8" PRIMARY KEY ("id"))
query: ALTER TABLE "LESSONS" ADD CONSTRAINT "FK_8d73f4e78c54a421244388fa951" FOREIGN KEY ("courseId") REFERENCES "COURSES"("id") ON DELETE NO ACTI
ON ON UPDATE NO ACTION
query: COMMIT
info: Data source initialized
info: Server started on port 3000
```

psql url and run \dt; (table created w 2 columns)

```
PS C:\Users\rohad\WebstormProjects\TS-Bootcamp> psql postgresql://rest_api_gx66_user:3dX5szHxYi55dYwUoBrGQvCMqmpvulGc@dpg-cv7i0q5umphs73c6f5j0-a.f
rankfurt-postgres.render.com/rest_api_gx66
psql (17.4, server 16.8 (Debian 16.8-1.pgdg120+1))
WARNING: Console code page (850) differs from Windows code page (1252)
         8-bit characters might not work correctly. See psql reference
         page "Notes for Windows users" for details.
SSL connection (protocol: TLSv1.3, cipher: TLS_AES_128_GCM_SHA256, compression: off, ALPN: none)
Type "help" for help.

rest_api_gx66=> \dt;
              List of relations
 Schema |  Name   | Type  |       Owner
--------+---------+-------+-------------------
 public | COURSES | table | rest_api_gx66_user
 public | LESSONS | table | rest_api_gx66_user
(2 rows)
```

40. populate db from local test data from db-data.ts, in package.json:

```
"populate-db": "npm-run-all clean build run-populate-db-script",
"run-populate-db-script": "node dist/models/populate-db.js"
```

populate-db.ts:

```
async function populateDb() {
  await AppDataSource.initialize();
  console.log("Data connection initialized");

  const courses = Object.values(COURSES) as DeepPartial<Course>[];

  const courseRepository = AppDataSource.getRepository(Course);
  const lessonRepository = AppDataSource.getRepository(Lesson);

  for (let courseData of courses) {
    console.log(`Inserting course ${courseData.title}`);

    const course = courseRepository.create(courseData);

    await courseRepository.save(course);

    for (let lessonData of courseData.lessons) {
      console.log(`Inserting lesson ${lessonData.title}`);

      const lesson = lessonRepository.create(lessonData);

      lesson.course = course;

      await lessonRepository.save(lesson);
    }
  }

  const totalCourses = await courseRepository.createQueryBuilder().getCount();
  const totalLessons = await lessonRepository.createQueryBuilder().getCount();

  console.log(
    `Database populated with ${totalCourses} courses and ${totalLessons} lessons`
  );
}

populateDb()
  .then(() => {
    console.log("Populating database with seed data");
    process.exit(0);
  })
```

41. deleting db script, package.json

```
"delete-db": "npm-run-all clean build run-delete-db-script",
"run-delete-db-script": "node dist/models/delete-db.js"
```

delete-db.ts

```
import * as dotenv from "dotenv";

const results = dotenv.config();

import "reflect-metadata";
import { AppDataSource } from "../data-source";
import { Lesson } from "./lesson";
import { Course } from "./course";


async function deleteDb() {
  await AppDataSource.initialize();
  console.log("Data connection initialized");

  console.log("Deleting lessons table");
  await AppDataSource.getRepository(Lesson).delete({});

  console.log("Deleting courses table");
  await AppDataSource.getRepository(Course).delete({});

  console.log("Database tables deleted, closing connection");
}

deleteDb()
  .then(() => {
    console.log("Database deleted");
    process.exit(0);
  })
  .catch((error) => {
    console.log("Error deleting database", error);
    process.exit(1);
  });
```

connect psql to check:
$ psql postgresql://rest_api_gx66_user:3dX5szHxYi55dYwUoBrGQvCMqmpvulGc@d6f5j0-a.frankfurt-postgres.render.com/rest_api_gx66

```
rest_api_gx66=> selece count(*) from "LESSONS";
ERROR:  syntax error at or near "selece"
LINE 1: selece count(*) from "LESSONS";
        ^
rest_api_gx66=> select count(*) from "LESSONS";
 count
-------
     0
(1 row)
```

after running populate-db script again:

```
rest_api_gx66=> select count(*) from "LESSONS";
 count
-------
   132
(1 row)
```

42. Data Retrieval Express Endpoint Using TypeORM
in roots folder new file get-all-courses.ts

```ts
import { Request, Response } from "express";
import { logger } from "../logger";
import { AppDataSource } from "../data-source";

export async function gettAllCourses(request: Request, response: Response) {
  logger.info("Getting all courses");

  const courses = await AppDataSource.getRepository("Course")
    .createQueryBuilder("courses")
    .orderBy("courses.seqNo")
    .getMany();

  response.status(200).json({ courses });
}
```

in server.ts addin route

```ts
function setupExpress() {
  app.route("/").get(root);

  app.route("/api/courses").get(gettAllCourses);
}
```

hitting endpoint (http://localhost:3000/api/courses):


```
{
  "courses": [
    {
      "id": 36,
      "seqNo": 0,
      "title": "Typescript Bootcamp",
      "longDescription": "Learn in depth the Typescript language, build practical real-world projects",
      "category": "BEGINNER",
      "createdDate": "2025-03-13T06:41:12.053Z",
      "lastModifiedDate": "2025-03-13T06:41:12.053Z"
    },
    {
      "id": 29,
      "seqNo": 1,
      "title": "Angular Material Course",
      "longDescription": "Build Applications with the official Angular UI Widget Library",
      "category": "BEGINNER",
      "createdDate": "2025-03-13T06:41:00.975Z",
      "lastModifiedDate": "2025-03-13T06:41:00.975Z"
    },
```


43. Bidirectional relationship between Course and Lesson (one to many, many to one)
Adding leftJoinAndSelect by typeORM

```ts
import { Request, Response } from "express";
import { logger } from "../logger";
import { AppDataSource } from "../data-source";

export async function gettAllCourses(request: Request, response: Response) {
  logger.info("Getting all courses");

  const courses = await AppDataSource.getRepository("Course")
    .createQueryBuilder("courses")
    .leftJoinAndSelect("courses.lessons", "LESSONS")
    .orderBy("courses.seqNo")
    .getMany();

  response.status(200).json({ courses });
}
```

44. Error handling; NextFunction from express is a callback that you call to pass control to the next middleware function in the request-response cycle.

- Send a response (`res.send`, `res.json`, etc.).
- Pass control to the next middleware function by calling `next()`.

```
export async function gettAllCourses(
  request: Request,
  response: Response,
  next: NextFunction
) {
  logger.info("Getting all courses");

  try {
    const courses = await AppDataSource.getRepository("Course")
      .createQueryBuilder("courses")
      .orderBy("courses.seqNo")
      .getMany();

    response.status(200).json({ courses });
  } catch (error) {
    logger.error("Error getting all courses", error);
    return next(error);
  }
}
```

45. Overwrite express default error handling by creating our own middleware

In our case, what we want to add to our server is a default error handling middleware that is going to be used in case that everything else has failed. So this middleware is going to be the last link in the middleware chain. If all the previous middleware did not manage to generate a valid HTTP response, then in that case, the default error handling middleware is going to handle the error and write back an error response to the client. Middleware in Express.js is a function that runs between the request and response cycle. It can modify the request (req) and response (res) objects or execute additional logic before passing control to the next middleware using next().

- creating middleware folder and default-error-handling.ts

```
import { Request, Response, NextFunction } from "express";
import { logger } from "../logger";

export function defaultErrorHandler(
  error, //its only filled in case of an error
  request: Request,
  response: Response,
  next: NextFunction
) {
  logger.error("Error in request", error);

  if (response.headersSent) {
    logger.error(
      "Response headers already sent, delegating to built-in Express default error handl
    );
    return next(error);
  }

  response.status(500).json({
    status: "error",
    message: "Default error handling triggered, check logs for more details",
  });
}
```

- plug in to the chain of middlewares (last place!) in server.ts

  So now we are sure that if something goes wrong in production with our rest endpoints, at least we're going to be able to catch the error and view it properly in the server logs.

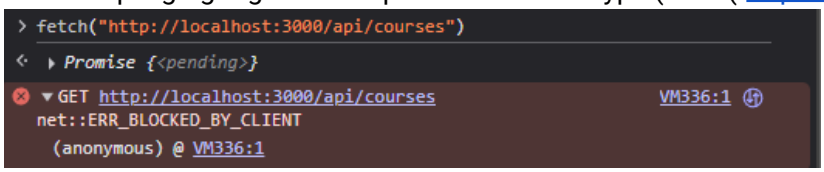
```
import { defaultErrorHandler } from "./middlewares/default-error-handling";
```

```
function setupExpress() {
  app.route("/").get(root);

  app.route("/api/courses").get(gettAllCourses);

  app.use(defaultErrorHandler);
}
```

46. CORS capabilities (Cross-Origin Resource Sharing)

CORS is a security feature implemented by web browsers that controls how resources on a web page can be requested from another domain. It prevents unauthorized cross-origin requests, helping to protect users from malicious sites trying to access sensitive data.

For example go google.co.uk open console and type (fetch("http://localhost:3000/api/courses"):



- By default, browsers enforce the **Same-Origin Policy (SOP)**, meaning a web page can only request resources from the same domain.
- If a web page needs to fetch data from a different domain (cross-origin request), the server must explicitly allow it by including **CORS headers** in the response.

CORS Headers:

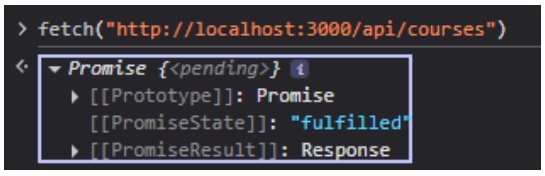
- CORS Headers:Access-Control-Allow-Origin: Specifies which origins can access the resource (e.g., * for all, or a specific domain).
- Access-Control-Allow-Methods: Lists HTTP methods (GET, POST, PUT, DELETE, etc.) allowed in cross-origin requests.
- Access-Control-Allow-Headers: Specifies which request headers are permitted.
- Access-Control-Allow-Credentials: Indicates whether cookies or authentication credentials are allowed.

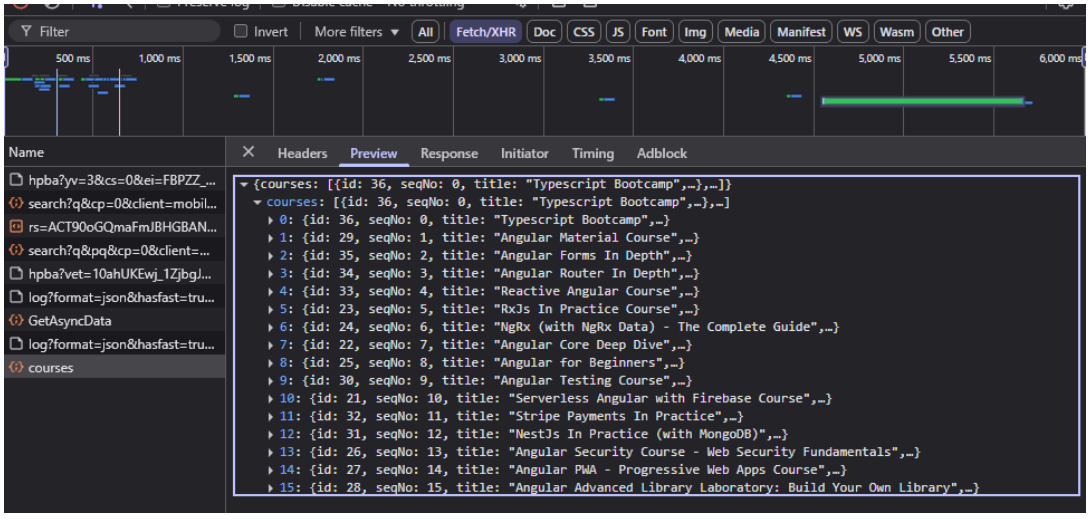
install CORS

$ npm i cors

using it in our server.ts (setting it "*", be accessible from anywhere)

```
const cors = require("cors");

const app = express();

function setupExpress() {
  app.use(cors("*"));

  app.route("/").get(root);

  app.route("/api/courses").get(gettAllCourses);

  app.use(defaultErrorHandler);
}
```





47. Filterins Query results TypeORM (new data retrieval endpoints)
Retrieve only one result by using the url for example:

```
19: {
    id: 19,
    title: 'Angular Forms In Depth',
    longDescription: 'Build complex enterprise data forms with the powerful Angular Forms module',
    iconUrl: 'https://angular-university.s3-us-west-1.amazonaws.com/course-images/angular-forms-course-small.jpg',
    category: 'BEGINNER',
    lessons: [],
    seqNo: 2,
    url: 'angular-forms-course',
    price: 50
},
```

- remove database: **$ npm run delete-db**
- Adding new property to our model in models/course.ts

```
@Column()
url: string;
```

- Running in synchronized mode using NPM run dev. And this is going to update the schema in the database according with the model **$ npm run dev**
- Populating db w the new schema: **$ npm run populate-db**

In routes folder we add a new file called find-course-by-url.ts

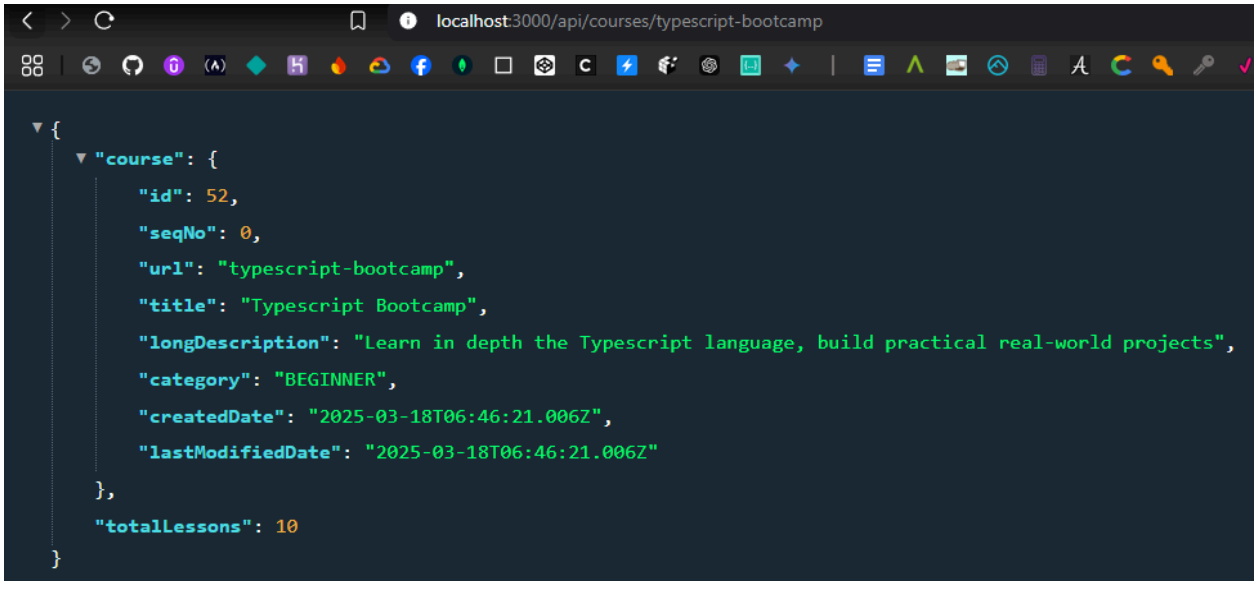
```
export async function findCourseByUrl(
  request: Request,
  response: Response,
  next: NextFunction
) {
  try {
    logger.debug("Called findCourseByUrl()");

    const courseUrl = request.params.courseUrl;

    if (!courseUrl) {
      throw "Missing courseUrl parameter";
    }
  } catch (error) {
    logger.error("Error calling findCourseByUrl", error);
    return next(error);
  }
}
```

plug it in in server.ts

```
function setupExpress() {
  app.use(cors("*"));

  app.route("/").get(root);

  app.route("/api/courses").get(gettAllCourses);

  app.route("/api/courses/:courseUrl").get(findCourseByUrl);

  app.use(defaultErrorHandler);
}
```

setting query up in find-course-by-url.ts

```
export async function findCourseByUrl(
  request: Request,
  response: Response,
  next: NextFunction
) {
  try {
    logger.debug("Called findCourseByUrl()");

    const courseUrl = request.params.courseUrl;

    if (!courseUrl) {
      throw "Missing courseUrl parameter";
    }

    const course = await AppDataSource.getRepository("Course").findOneBy({
      url: courseUrl,
    });

    if (!course) {
      const message = `Could not find course with url ${courseUrl}`;
      logger.error(message);
      response.status(404).json({ message });
      return;
    }

    const totalLessons = await AppDataSource.getRepository(Lesson)
      .createQueryBuilder("lessons")
      .where("lessons.courseId = :courseId", { courseId: course.id })
      .getCount();

    response.status(200).json({ course, totalLessons });
  } catch (error) {
    logger.error("Error calling findCourseByUrl", error);
    return next(error);
  }
}
```



48. Implementing find course lessons endpoint

In roots folder new file: find-lesson-for-course.ts

```ts
import { Request, Response, NextFunction } from "express";
import { logger } from "../logger";

export async function findLessonForCourse(
  request: Request,
  response: Response,
  next: NextFunction
) {
  try {
    logger.debug("Called findlessonForCourse()");
  } catch (error) {
    logger.error("Error calling findlessonForCourse()", error);
    return next(error);
  }
}
```

server.ts

```ts
function setupExpress() {
  app.use(cors("*"));

  app.route("/").get(root);

  app.route("/api/courses").get(gettAllCourses);

  app.route("/api/courses/:courseUrl").get(findCourseByUrl);

  app.route("/api/courses/:courseId/lessons").get(findLessonForCourse);

  app.use(defaultErrorHandler);
}
```

find-lesson-for-course.ts

```ts
export async function findLessonForCourse(
  request: Request,
  response: Response,
  next: NextFunction
) {
  try {
    logger.debug("Called findlessonForCourse()");

    const courseId = request.params.courseId,
      query = request.query as any,
      pageNumber = query?.pageNumber ?? 0,
      pageSize = query?.pageSize ?? 3;
    // Page number and page size are optional parameters.
    // If we don't pass in a page number with our call to this end point,
    // we should assume that the page number is zero, meaning the first page of data.
    // We can do that by using the optional shading syntax. See above

    if (!isInteger(courseId)) {
      throw `Invalid courseId: ${courseId}`;
    }

    if (!isInteger(pageNumber)) {
      throw `Invalid pageNumber: ${pageNumber}`;
    }

    if (!isInteger(pageSize)) {
      throw `Invalid pageSize: ${pageSize}`;
    }
  } catch (error) {
    logger.error("Error calling findlessonForCourse()", error);
    return next(error);
  }
}
```

49. Finishing find course lessons endpoint With TypeORM

```ts
export async function findLessonForCourse(
  request: Request,
  response: Response,
  next: NextFunction
) {
  try {
    logger.debug("Called findlessonForCourse()");

    const courseId = request.params.courseId,
      query = request.query as any,
      pageNumber = query?.pageNumber ?? "0",
      pageSize = query?.pageSize ?? "3";

    // Page number and page size are optional parameters.

    if (!isInteger(courseId)) {
      throw `Invalid courseId: ${courseId}`;
    }

    if (!isInteger(pageNumber)) {
      throw `Invalid pageNumber: ${pageNumber}`;
    }

    if (!isInteger(pageSize)) {
      throw `Invalid pageSize: ${pageSize}`;
    }

    const lessons = await AppDataSource.getRepository(Lesson)
      .createQueryBuilder("lessons")
      .where("lessons.courseId = :courseId", { courseId })
      .orderBy("lessons.seqNo")
      .skip(pageNumber * pageSize)
      .take(pageSize)
      .getMany();

    response.status(200).json({ lessons });
  } catch (error) {
    logger.error("Error calling findlessonForCourse()", error);
    return next(error);
  }
}
```


```json
{
  "lessons": [
    {
      "id": 131,
      "title": "Setting Up the Development Environment",
      "duration": "0:44",
      "seqNo": 1,
      "createdDate": "2025-03-18T06:46:21.470Z",
      "lastModifiedDate": "2025-03-18T06:46:21.470Z"
    },
    {
      "id": 132,
      "title": "Why Typescript? Key Benefits of the Language",
      "duration": "12:33",
      "seqNo": 2,
      "createdDate": "2025-03-18T06:46:21.583Z",
      "lastModifiedDate": "2025-03-18T06:46:21.583Z"
    },
    {
      "id": 133,
      "title": "Compiling Your First Typescript Program",
      "duration": "05:18",
      "seqNo": 3,
      "createdDate": "2025-03-18T06:46:21.707Z",
      "lastModifiedDate": "2025-03-18T06:46:21.707Z"
    }
  ]
}
```

50. Data modification endpoints (json body parser)

New route update-course.ts

```
export async function updateCourse(
  request: Request,
  response: Response,
  next: NextFunction
) {
  try {
    logger.debug("Called updateCourse()");
  } catch (error) {
    logger.error("Error calling updateCourse()", error);
    return next(error);
  }
}
```

server.ts new endpoint

```
app.route("/api/courses/:courseId").patch(updateCourse);
```

install body-parser express middleware
$ npm i body-parser

add it to server.ts so parsing the body will accept json

```
const cors = require("cors");

const bodyParser = require("body-parser");

const app = express();

function setupExpress() {
  app.use(cors("*"));

  app.use(bodyParser.json()); // for parsing application/json

  app.route("/").get(root);

  app.route("/api/courses").get(gettAllCourses);

  app.route("/api/courses/:courseUrl").get(findCourseByUrl);

  app.route("/api/courses/:courseId/lessons").get(findLessonForCourse);

  app.route("api/courses/:courseId").patch(updateCourse);

  app.use(defaultErrorHandler);
}
```

Finishing update-course.ts

```ts
import { Request, Response, NextFunction } from "express";
import { logger } from "../logger";
import { isInteger } from "../utils";
import { AppDataSource } from "../data-source";
import { Course } from "../models/course";

export async function updateCourse(
  request: Request,
  response: Response,
  next: NextFunction
) {
  try {
    logger.debug("Called updateCourse()");

    const courseId = request.params.courseId,
      changes = request.body;

    if (!isInteger(courseId)) {
      throw `Invalid courseId: ${courseId}`;
    }

    await AppDataSource.createQueryBuilder()
      .update(Course)
      .set(changes)
      .where("id = :courseId", { courseId })
      .execute();
    response
      .status(200)
      .json({ message: `Course ${courseId} was updated successfully` });
  } catch (error) {
    logger.error("Error calling updateCourse()", error);
    return next(error);
  }
}
```

51. testing data modification endpoint using CURL (Client URL)
A command-line tool used to transfer data from or to a server using various protocols like HTTP, HTTPS, FTP, and more. It's commonly used for making API requests, downloading files, and testing web services.

$ curl -X PATCH "http://localhost:3000/api/courses/52" -H "Content-Type: application/json" -d '{"title":"Typescript Bootcamp v2"}'

- curl → Command-line tool to make HTTP requests
- -X PATCH → Use the PATCH method (for partial updates)
- "http://localhost:3000/api/courses/52" → The API endpoint (updating course with ID 52)
- -H "Content-Type: application/json" → Sets request header to JSON format
- -d '{"title":"Typescript Bootcamp v2"}' → Sends this JSON data in the request body

```
$ curl -X PATCH "http://localhost:3000/api/courses/52" -H "Content-Type: application/json" -d '{"title":"Typescript Bootcamp v2"}'
{"message":"Course 52 was updated successfully"}
```


```
▼ "courses": [
  ▼ {
      "id": 52,
      "seqNo": 0,
      "url": "typescript-bootcamp",
      "title": "Typescript Bootcamp v2",
```

52. Create Course endpoint w TypeORM transaction manager

new file in routes folder: create-course.ts

```
export async function createCourse(
  request: Request,
  response: Response,
  next: NextFunction
) {
  try {
    logger.debug("Called createCourse()");
    const newCourse = request.body;
  } catch (error) {
    logger.error(`Error calling createCourse(), ${error}`);
    return next(error);
  }
}
```

new route in server.ts

```
app.route("/api/courses").post(createCourse);
```

continue in create-course.ts

All courses has to have a unique seqNum, in order to prevent happening two courses having the same seqNum (or get submitted the same time by 2 users) we need to set the isolation level which means that those two queries need to run in the same transaction, and that database transaction needs to run in the repeatable read isolation mode, meaning that any reads from the database need to be repeatable as long as the transaction is still ongoing and it has not yet either been committed or rolled back.

seqNumber will be the maximum plus one and until the transaction completes it won't be saved, neither other can require the same number as they will need to wait until it's saved (repeatable read).

Transactions (SQL Isolation Level):

Isolation Level	Prevents Dirty Reads	Prevents Non-Repeatable Reads	Prevents Phantom Reads
READ UNCOMMITTED	❌ No	❌ No	❌ No
READ COMMITTED	✅ Yes	❌ No	❌ No
REPEATABLE READ	✅ Yes	✅ Yes	❌ No
SERIALIZABLE	✅ Yes	✅ Yes	✅ Yes

- Logs that `createCourse()` was called
- Extracts course data from the request body.
- Checks if data is missing and throws an error if so.
- Starts a database transaction with `REPEATABLE READ` for consistency.
- Retrieves the highest `seqNo` from the `courses` table.
- Creates a new course with an incremented `seqNo`.
- Saves the new course to the database.
- Returns the created course as a JSON response

```
export async function createCourse(
  request: Request,
  response: Response,
  next: NextFunction
) {
  try {
    logger.debug("Called createCourse()");
    const data = request.body;

    if (!data) {
      throw "Missing course data";
    }
    const course = await AppDataSource.manager.transaction(
      "REPEATABLE READ",
      async (transactionalEntityManager) => {
        //just to make the code more readable below
        const repository = transactionalEntityManager.getRepository(Course);

        const result = await repository
          .createQueryBuilder("courses")
          .select("MAX(courses.seqNo)", "max")
          .getRawOne();

        const course = repository.create({
          ...data,
          seqNo: (result?.max ?? 0) + 1,
        });
        await repository.save(course);

        return course;
      }
    );
    response.status(200).json({ course });
```

testing with Curl:
$ curl -X POST http://localhost:3000/api/courses -H "Content-Type: application/json" -d '{"url": "typescript-bootcamp TEST", "title": "Typescript Bootcamp TEST", "longDescription": "Testing post request", "category": "TEST"}'

```
debug: Called createCourse()
query: START TRANSACTION
query: SET TRANSACTION ISOLATION LEVEL REPEATABLE READ
query: SELECT MAX("courses"."seqNo") AS "max" FROM "COURSES" "courses"
query: INSERT INTO "COURSES"("seqNo", "url", "title", "longDescription", "category", "createdDate", "lastModifiedDate") VALUES ($1, $2,
otcamp TEST","Typescript Bootcamp TEST","Testing post request","TEST"]
query: COMMIT
```


```
{
    "id": 53,
    "seqNo": 16,
    "url": "typescript-bootcamp TEST",
    "title": "Typescript Bootcamp TEST",
    "longDescription": "Testing post request",
    "category": "TEST",
    "createdDate": "2025-03-25T06:47:38.315Z",
    "lastModifiedDate": "2025-03-25T06:47:38.315Z"
}
```

53. Delete Course and Lessons Endpoint

new file in routes folder delete-course.ts

```ts
import { NextFunction, Request, Response } from "express";
import { logger } from "../logger";

export async function deleteCourseAndLessons(
  request: Request,
  response: Response,
  next: NextFunction
) {
  try {
    logger.debug("Called deleteCourseAndLessons()");
  } catch (error) {
    logger.error("Error calling deleteCourseAndLessons()", error);
    return next(error);
  }
}
```

server.ts

```ts
app.route("/api/courses/:courseId").delete(deleteCourseAndLessons);
```

delete-course.ts

```ts
export async function deleteCourseAndLessons(
  request: Request,
  response: Response,
  next: NextFunction
) {
  try {
    logger.debug("Called deleteCourseAndLessons()");

    const courseId = request.params.courseId;

    if (!isInteger(courseId)) {
      throw `Invalid courseId: ${courseId}`;
    }

    AppDataSource.manager.transaction(async (transactionalEntityManager) => {
      await transactionalEntityManager
        .createQueryBuilder()
        .delete()
        .from(Lesson)
        .where("courseId = :courseId", { courseId })
        .execute();

      await transactionalEntityManager
        .createQueryBuilder()
        .delete()
        .from(Course)
        .where("id = :courseId", { courseId })
        .execute();
    });
    response
      .status(200)
      .json({ message: `Course ${courseId} was deleted successfully` });
  } catch (error) {
    logger.error("Error calling deleteCourseAndLessons()", error);
    return next(error);
  }
}
```

test w curl w bash

$ curl -X DELETE http://localhost:3000/api/courses/53

```
$ curl -X DELETE http://localhost:3000/api/courses/53
{"message":"Course 53 was deleted successfully"}
```

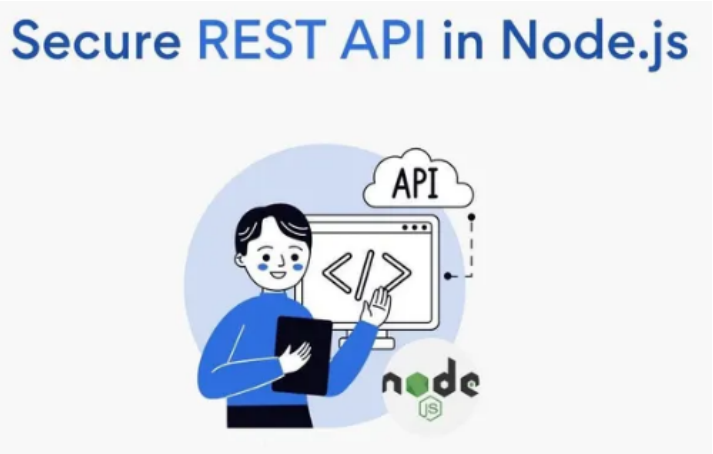
in node

```
debug: Called deleteCourseAndLessons()
query: START TRANSACTION
query: DELETE FROM "LESSONS" WHERE "courseId" = $1 -- PARAMETERS: ["53"]
query: DELETE FROM "COURSES" WHERE "id" = $1 -- PARAMETERS: ["53"]
query: COMMIT
```

checking in psql to make sure all the lessons are deleted too

$ psql postgresql://rest_api_gx66_user:@dpg-cvi073c5j0-a.frankfurt-postgres.render.com/rest_api_gx66

```
rest_api_gx66=> select * from "LESSONS" where "courseId" = 53;
 id | title | duration | seqNo | createdDate | lastModifiedDate | courseId
----+-------+----------+-------+-------------+------------------+----------
(0 rows)
```



To secure endpoints, we'll use authentication and authorization based on JSON Web Tokens (JWT). Each incoming request, such as creating a user or updating a course, will include a header containing a cryptographically signed JWT. When the server receives the request, it will validate the token to confirm the identity of the user making the request.

1. Adding users entity our modal and include in the data source

```typescript
import {
  Column,
  CreateDateColumn,
  Entity,
  PrimaryGeneratedColumn,
  UpdateDateColumn,
} from "typeorm";

@Entity({
  name: "USERS",
})
export class User {
  @PrimaryGeneratedColumn
  id: number;

  @Column()
  email: string;

  @Column()
  passwordHash: string;

  @Column()
  passwordSalt: string;

  @Column()
  pictureUrl: string;

  @Column()
  isAdmin: boolean;

  @CreateDateColumn()
  createdDate: Date;

  @UpdateDateColumn()
  lastModifiedDate: Date;
}
```

src > TS data-source.ts > AppDataSource

```typescript
import { DataSource } from "typeorm";
import { Course } from "./models/course";
import { Lesson } from "./models/lesson";
import { User } from "./models/user";

export const AppDataSource = new DataSource({
  type: "postgres",
  host: process.env.DB_HOST,
  username: process.env.DB_USERNAME,
  password: process.env.DB_PASSWORD,
  port: parseInt(process.env.DB_PORT),
  database: process.env.DB_NAME,
  ssl: true,
  entities: [Course, Lesson, User],
  synchronize: true,
  logging: true,
});
```

2. Update populatedb script

```
async function populateDb() {

  for (let courseData of courses) {···
  }

  const users = Object.values(USERS) as any[];

  for (let userData of users) {
    console.log(`Inserting user ${userData.email}`);
    const { email, pictureUrl, isAdmin, passwordSalt, plainTextPassword } =
      userData;

    AppDataSource.getRepository(User).create({
      email,
      pictureUrl,
      isAdmin,
      passwordSalt,
    });
  }
```

3. crypto.pbkdf2 - Storing plaintext passwords securely in db using node crypto module for hashing

- **Purpose:** Generates a cryptographic key from a password using the PBKDF2 (Password-Based Key Derivation Function 2) algorithm.
- **Use Case:** Commonly used for **password hashing and key generation** in secure applications.

Steps:
- Takes a **password**, a **salt** (random data for uniqueness), **iterations** (repetition count), **key length**, and a **hash algorithm** (e.g., SHA-256).
- Hashes the password and salt repeatedly based on the iteration count.
- Produces a **derived key** of the specified length, making brute-force attacks harder.

creating a new utility function
This utility function is convenient to convert functions that have a callback based API into a promise based API. (crypto.pbkdf2 is callback based we turn it to promise based using this provided by node)

- `crypto`: Node.js built-in module for cryptography (encryption, hashing, etc.).
- `util.promisify`: Converts callback-based functions into **Promise-based functions** for easier use with `async/await`.
- `crypto.pbkdf2`: Hashing function using PBKDF2 (Password-Based Key Derivation Function 2).
- `hashPassword`: Now a **Promise-based** version of `crypto.pbkdf2` (returns a Promise instead of using a callback).

```
const crypto = require("crypto");
const util = require("util");

const hashPassword = util.promisify(crypto.pbkdf2);
```

```typescript
export async function calculatePasswordHash(
  plainTextPassword: string,
  passwordSalt: string
) {
  const passwordhash = await hashPassword(
    plainTextPassword,
    passwordSalt,
    1000,
    64,
    "sha512"
  );

  return passwordhash.toString("hex");
}
```

Parameters:

- `plainTextPassword`: The raw password to hash.
- `passwordSalt`: A random, unique value (prevents identical hashes for identical passwords).

Arguments Passed to `hashPassword`:

1. `plainTextPassword`: The password to hash.
2. `passwordSalt`: The salt value (adds randomness).
3. `1000`: Number of iterations (how many times hashing is repeated for extra security).
4. `64`: Length of the derived key (64 bytes).
5. `"sha512"`: Hashing algorithm used (stronger than SHA-256).

Return Value:

- `passwordhash` is a Buffer (binary data).
- `.toString("hex")` converts it to a hexadecimal string for readability and storage.

Import our new utility function to populatedb and awaiting for its execution

```typescript
const users = Object.values(USERS) as any[];

for (let userData of users) {
  console.log(`Inserting user ${userData.email}`);
  const { email, pictureUrl, isAdmin, passwordSalt, plainTextPassword } =
    userData;

  const user = AppDataSource.getRepository(User).create({
    email,
    pictureUrl,
    isAdmin,
    passwordSalt,
    passwordHash: await calculatePasswordHash(
      plainTextPassword,
      passwordSalt
    ),
  });

  await AppDataSource.manager.save(user);
}
```

4. Update delete db script with users table included (we will need to use it several times later on) and populate it again

```
async function deleteDb() {
  await AppDataSource.initialize();
  console.log("Data connection initialized");

  console.log("Deleting LESSONS table");
  await AppDataSource.getRepository(Lesson).delete({});

  console.log("Deleting COURSES table");
  await AppDataSource.getRepository(Course).delete({});

  console.log("Deleting USERS table");
  await AppDataSource.getRepository(User).delete({});

  console.log("Database tables deleted, closing connection");
}
```

$ npm run delete-db

```
PS C:\Users\rohad\WebstormProjects\TS-Bootcamp\rest-api> npm run delete-db

> rest-api@1.0.0 delete-db
> npm-run-all clean build run-delete-db-script


> rest-api@1.0.0 clean
> rimraf dist


> rest-api@1.0.0 build
> tsc


> rest-api@1.0.0 run-delete-db-script
> node dist/models/delete-db.js
```

```
Data connection initialized
Deleting LESSONS table
query: DELETE FROM "LESSONS"
Deleting COURSES table
query: DELETE FROM "COURSES"
Deleting USERS table
query: DELETE FROM "USERS"
Database tables deleted, closing connection
Database deleted
```

$ npm run populate-db

```
Populating database with seed data
```

```
query: INSERT INTO "USERS"("email", "passwordHash", "passwordSalt", "pic
tureUrl", "isAdmin", "createdDate", "lastModifiedDate") VALUES ($1, $2,
$3, $4, $5, DEFAULT, DEFAULT) RETURNING "id", "createdDate", "lastModifi
edDate" -- PARAMETERS: ["test@angular-university.io","8444e1207312309e8a
0687bee0015633a1d51135ef75175d48786629e44e697be93a29ba70898f2c74107c3e82
7e6936346d3b0006a1354125a65ca755a691d6","o61TA7yaJIsa","https://angular-
academy.s3.amazonaws.com/main-logo/main-page-logo-small-hat.png",0]
query: COMMIT
Inserting user admin@angular-university.io
query: START TRANSACTION
query: INSERT INTO "USERS"("email", "passwordHash", "passwordSalt", "pic
tureUrl", "isAdmin", "createdDate", "lastModifiedDate") VALUES ($1, $2,
$3, $4, $5, DEFAULT, DEFAULT) RETURNING "id", "createdDate", "lastModifi
edDate" -- PARAMETERS: ["admin@angular-university.io","d8a817ee00b179019
3ce4d0f2824c81d5f042c55e91dbaa458c70ae15f5fe9bb02943ddfb33117c658ef5ec63
9d75cb4e4ba02bd28546b83340c25c77a5d9b07","NydKRjIh4T4X","https://angular
-academy.s3.amazonaws.com/main-logo/main-page-logo-small-hat.png",1]
query: COMMIT
```

5. Create user endpoint

server.ts

```
app.route("/api/users").post(createUser);
```

create-user.ts

```
const crypto = require("crypto");

export async function createUser(
  request: Request,
  response: Response,
  next: NextFunction
) {
  try {
    logger.debug(`Called createUser()`);

    const { email, password, pictureUrl, isAdmin } = request.body;

    if (!email) {
      throw "Could not extract email from request, aborting";
    }
    if (!password) {
      throw "Could not extract password from request, aborting";
    }

    const repository = AppDataSource.getRepository(User);

    const user = await repository
      .createQueryBuilder("users")
      .where("email = :email", { email })
      .getOne();

    if (user) {
      const message = `User with email ${email} already exists`;
      logger.error(message);
      response.status(500).json({ message });
      return;
    }

    const passwordSalt = crypto.randomBytes(64).toString("hex");
    const passwordHash = await calculatePasswordHash(password, passwordSalt);

    const newUser = repository.create({
      email,
      pictureUrl,
      isAdmin,
      passwordSalt,
      passwordHash,
    });

    AppDataSource.manager.save(newUser).then(() => {
      logger.debug(`User ${email} created`);
      response.status(200).json({ message: "User created" });
    });
  } catch (anyErr) {
```

test w curl (in gitbash):

$ curl -X POST http://localhost:3000/api/users -H "Content-Type: application/json" -d '{"email": "asd@asdasd.com", "pictureUrl": "https://picturl.com", "password" : "testpassword", "isAdmin" : "false"}'

gitbash: `{"message":"User created"}` node: `debug: User asd@asdasd.com created`

6. Login endpoint

server.ts

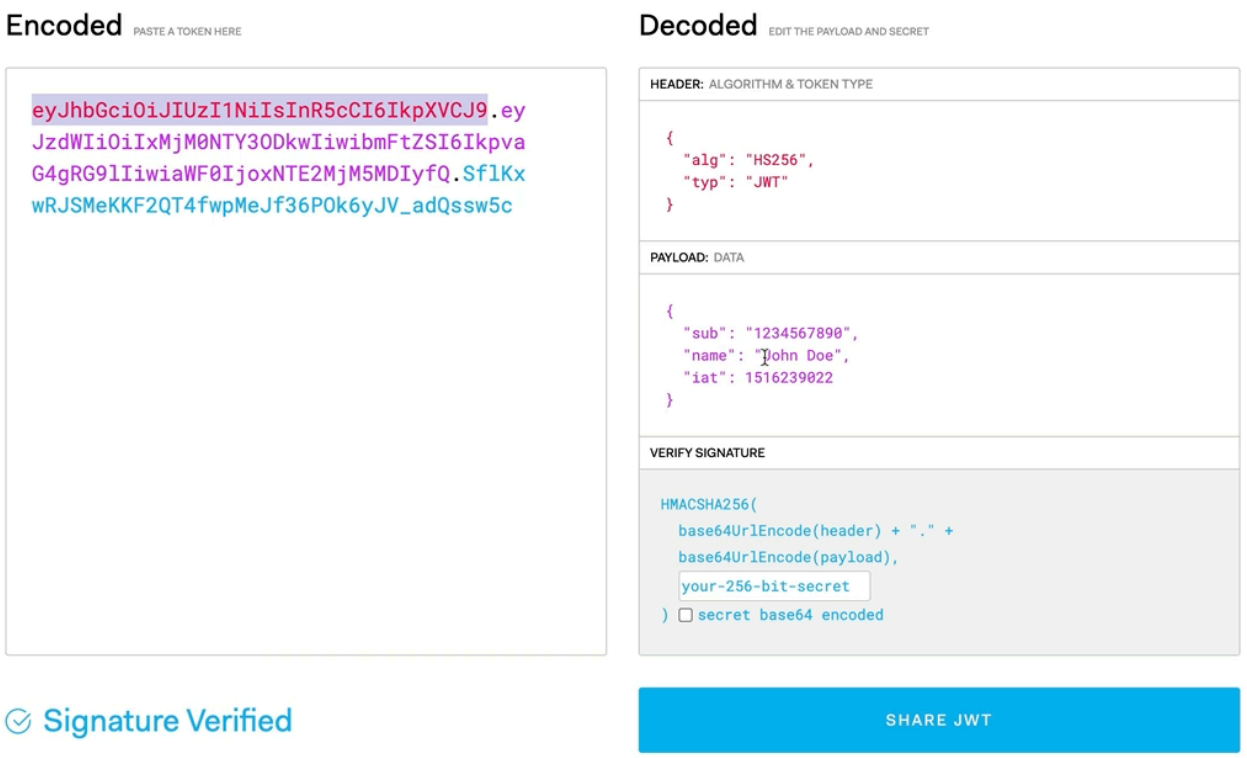
```
app.route("/api/login").post(login);
```

login.ts (skeleton without JWT)

```
export async function login(
  request: Request,
  response: Response,
  next: NextFunction
) {
  try {
    logger.debug(`Called login()`);
    const { email, password } = request.body;

    if (!email) {
      throw "Could not extract email from request, aborting";
    }
    if (!password) {
      throw "Could not extract password from request, aborting";
    }

    const user = await AppDataSource.getRepository("User")
      .createQueryBuilder("users")
      .where("email = :email", { email })
      .getOne();

    if (!user) {
      const message = "Login denied";
      logger.info(`${message} - ${email}`);
      response.status(403).json({ message });
      return;
    }

    const passwordHash = await calculatePasswordHash(
      password,
      user.passwordSalt
    );

    if (passwordHash !== user.passwordHash) {
      const message = "Login denied";
      logger.info(
        `${message} - user with ${email} email entered wrong password`
      );
      response.status(403).json({ message });
      return;
    }

    logger.info(`Login granted for user with ${email} email`);

    const { pictureUrl, isAdmin } = user;

    response.status(200).json({ user: { email, pictureUrl, isAdmin } });
```

7. JSON Web Token (JWT)

JWT (JSON Web Token) is a secure way to transmit information between parties as a JSON object. It's commonly used for authentication and authorization.



How JWT Works:

- **Structure:** A JWT has three parts separated by dots:
  - **Header:** Contains the algorithm and token type (e.g., HS256, JWT).
  - **Payload:** Contains user data/claims (e.g., user ID, roles).
  - **Signature:** Verifies the token wasn't tampered with, using a secret key.
- **Process:**
  - The server generates a JWT when a user logs in successfully.
  - The token is sent back to the client and stored (e.g., in localStorage or cookies).
  - For each request to a protected route, the token is sent in the HTTP header.
  - The server verifies the token's signature and checks user permissions.
- **Benefits:**
  - Stateless (no need for server-side session storage)
  - Secure if used correctly
  - Portable and easy to handle
- **Drawbacks:**
  - Once issued, it can't be easily revoked until expiration.
  - Sensitive data in the payload isn't encrypted by default—only encoded.

One computer might be using UTF-8, another computer might be using something else. So you want a way to be able to send strings across computers without necessarily knowing what is the encoding used by the target computer. So how do we solve this? Well, we use base 64. **Base 64** uses 64 characters carefully chosen so that they are represented the exact same way in all across the most common encoding used by computers.

8. Add JWT to login endpoint

$ npm install jsonwebtoken

create a unique secret key
start node: **$ node** -> **$ const crypto = require('crypto')** -> **$ crypto.randomBytes(32).toString("hex")**

```
'3bef7dd3a0f972845a46b5b63528a91db0404344f04ed4d74d55c8968d65eb84'
```

add it to our .env

```
JWT_SECRET=3bef7dd3a0f972845a46b5b63528a91db0404344f04ed4d74d55c8968d65eb84
```

use it in our login endpoint and send it back aspayload

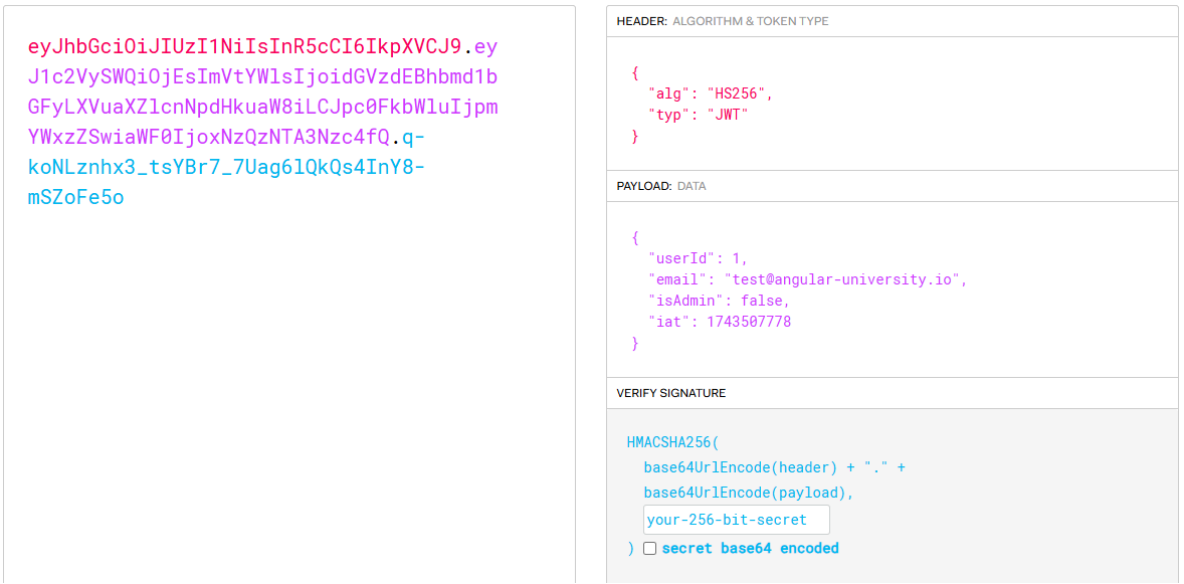
```
const JWT_SECRET = process.env.JWT_SECRET;
const jwt = require("jsonwebtoken");
```

```
const authJwtToken = await jwt.sign(authJwt, JWT_SECRET);

response
  .status(200)
  .json({ user: { email, pictureUrl, isAdmin }, authJwtToken });
```

test w curl
$ curl -X POST http://localhost:3000/api/login -H "Content-Type: application/json" -d '{"email": "test@angular-university.io", "password": "test"}'

```
$ curl -X POST http://localhost:3000/api/login -H "Content-Type: application/json" -d '{"email": "test@angular-univ
ersity.io", "password": "test"}'
{"user":{"email":"test@angular-university.io","pictureUrl":"https://angular-academy.s3.amazonaws.com/main-logo/main
-page-logo-small-hat.png","isAdmin":false},"authJwtToken":"eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJ1c2VySWQiOjEsImV
tYWlsIjoidGVzdEBhbmd1bGFyLXVuaXZlcnNpdHkuaW8iLCJpc0FkbWluIjpmYWxzZSwiaWF0IjoxNzQzNTA3Nzc4fQ.q-koNLznhx3_tsYBr7_7Uag
6lQkQs4InY8-mSZoFe5o"}
```

https://jwt.io/ pasting created token



9. Designing an Express Authentication Middleware

   Your server has multiple data retrieval and modification endpoints, some (all apart from login) of which require admin privileges based on a flag in the database. To handle authentication, the login endpoint accepts an email and password, returning a JSON Web Token (JWT).
   The client must store the JWT and include it in the Authorization header for protected endpoints like creating or updating courses. The server verifies the JWT's validity before granting access.
   To avoid repeating authentication logic in each route, the authentication checks will be centralized in a middleware function, ensuring all endpoints (except login) require valid tokens. Unauthorized requests will return a 403 Forbidden status.

   New middleware, creating authentication_middleware.ts in middlewares folder

10. **Token Extraction:**
    a. Retrieves the JWT from the `Authorization` header (`request.headers.authorization`).
    b. If no token is present, it logs a message and sends a **403 Forbidden** response.
11. **JWT Verification:**
    a. Calls `checkJwtValidity` to validate the token asynchronously using `jwt.verify` with the secret key `JWT_SECRET` from environment variables.
    b. If verification succeeds, the decoded user data is attached to the request object (`request["user"]`), and `next()` is called to proceed.
    c. If verification fails or the token is invalid, it logs an error and responds with **403 Forbidden**.
12. **checkJwtValidity Function:**
    a. Asynchronously verifies the JWT and returns the decoded user data if valid.
    b. Logs the user details found in the token.

```
const jwt = require("jsonwebtoken");
const JWT_SECRET = process.env.JWT_SECRET;

export function checkIfAuthenticated(
  request: Request,
  response: Response,
  next: NextFunction
) {
  const authJwtToken = request.headers.authorization;

  if (!authJwtToken) {
    logger.info("The JWT token not present, access denied");
    response.sendStatus(403);
    return;
  }

  checkJwtValidity(authJwtToken)
    .then((user) => {
      logger.info(`Authentication JWT sucessfully decoded:`, user);
      request["user"] = user;
      next();
    })
    .catch((error) => {
      logger.error("Error while checking JWT token", error);
      response.sendStatus(403);
    });
}

async function checkJwtValidity(authJwtToken: string) {
  const user = await jwt.verify(authJwtToken, JWT_SECRET);

  logger.info("Found user details in JWT token:", user);
  return user;
}
```

13. Test w CURL

without authorization header
$ curl -X GET http://localhost:3000/api/courses

gitbash:

```
$ curl -X GET http://localhost:3000/api/courses
Forbidden
```

node:

```
info: The JWT token not present, access denied
```

with authorization header
$ curl -X GET http://localhost:3000/api/courses -H "Authorization: eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJ1c2VySWQiOjEsImVtYWlsIjoidGVzdEBhbmd1bGFyLXVuaXZlcnNpdHkuaW8iLCJpc0FkbWluIjpmYWxzZSwiaWF0IjoxNzQzNTkyNTc0fQ.ZGjXbemPDF_qZDeupaN2XBfjwM2aP_LEe5-5wyLDxRo"

gitbash:

```
$ curl -X GET http://localhost:3000/api/courses -H "Authorization: eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJ1c2VySWQiOjEsImVtYWlsIjoidGVzdEBhbmd1bGFyLXVuaXZlcnNpdHkuaW8iLCJpc0FkbWluIjpmYWxzZSwiaWF0IjoxNzQzNTkyNTc0fQ.ZGjXbemPDF_qZDeupaN2XBfjwM2aP_LEe5-5wyLDxRo"
{"courses":[{"id":69,"seqNo":0,"url":"typescript-bootcamp","title":"Typescript Bootcamp","longDescription":"Learn in depth the Typescript language, build practical real-world projects","category":"BEGINNER","createdDate":"2025-03-28T07:00:24.710Z","lastModifiedDate":"2025-03-28T07:00:24.710Z"},{"id":62,"seqNo":1,"url":"angular-material-course","title":"Angular Material Course","longDescription":"Build Applications with the official Angular UI Widget Library","category":"BEGINNER","createdDate":"2025-03-28T07:00:10.921Z","lastModifiedDate":"2025-03-28T07:00:10.921Z"},{"id":68,"seqNo":2,"url":"angular-forms-course","title":"Angular Forms In Depth","longDescription":"Build c
```

node:

```
info: Found user details in JWT token: {"email":"test@angular-university.io","iat":1743592574,"isAdmin":false,"userId":1}
info: Authentication JWT sucessfully decoded: {"email":"test@angular-university.io","iat":1743592574,"isAdmin":false,"userId":1}
info: Getting all courses
query: SELECT "courses"."id" AS "courses_id", "courses"."seqNo" AS "courses_seqNo", "courses"."url" AS "courses_url", "courses"."title" AS "courses_title", "courses"."longDescription" AS "courses_longDescription", "courses"."category" AS "courses_category", "courses"."createdDate" AS "courses_createdDate", "courses"."lastModifiedDate" AS "courses_lastModifiedDate" FROM "COURSES" "courses" ORDER BY "courses"."seqNo" ASC
```

14. Admin only middleware, some endpoint requires admin flag (creating users)

server.ts

```
app.route("/api/users").post(checkIfAuthenticated, checkIfAdmin, createUser);
```

admin-only-middleware.ts

```
import { Request, Response, NextFunction } from "express";
import { logger } from "../logger";

export function checkIfAdmin(
  request: Request,
  response: Response,
  next: NextFunction
) {
  const user = request["user"];

  if (!user?.isAdmin) {
    logger.error("User is not admin, access denied");
    response.sendStatus(403);
    return;
  }

  logger.info("User is admin, access granted");
  next();
}
```

test w curl, admin JWT:

```
info: Authentication JWT successfully decoded: {"email":"test@angular-university.io","iat":1665755867,"isAdmin":false,"userId":1}
error: The user is not an admin, access denied
```

```
vasco@Vascos-iMac-2 typescript-bootcamp % curl -X POST http://localhost:9000/api/users -H "Content-Type:application/json" -d '{"email": "new-user-2@angular-university.io", "pictureUrl":"https://avatars.githubusercontent.com/u/5454709", "password": "test123", "isAdmin": false}' -H "Authorization: eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJ1c2VySWQiOjIsImVtYWlsIjoiYWRtaW5AYW5ndWxhci11bml2ZXJzaXR5LmlvIiwiaXNBZG1pbiI6dHJ1ZSwiaWF0IjoxNjY1NzU2MTcyfQ.V8f_yXIpDsBlvezbqkTVjms-TjFDJ4XtnpevkQgbwbs"
{"email":"new-user-2@angular-university.io","pictureUrl":"https://avatars.githubusercontent.com/u/5454709","isAdmin":false}%
```

```
info: User new-user-2@angular-university.io has been created.
```


Fin