TYPESCRIPT

FULLSTACK

TRAININIG

- Training goals and program
- Questions, discusion, expectations
- Elastic program contents

YOUR TRAINER:

MATEUSZ KULESZA

- Senior Software Developer,
- Team Leader, Scrum Master
- Project Manager
- Consultant and trainer

EXPERIENCE

Multiple years of commercial experience in:

- HTML5, CSS3, SVG, EcmaScript 5 i 6
- jQuery, underscore, backbone.js
- canjs, requirejs, dojo …
- Grunt, Gulp, Webpack, Karma, Jasmine ...
- Angular.JS, Angular2, React, RxJS, Flux
- NodeJS, Express, Typescript, MongoDB

TYPESCRIPT

TypeScript is a programming language with JS-based syntax.

It adds static and strong typing to enable us to write safer and more reliable code.

TS compiles primarily against JavaScript, but can also be used with WebAssembly (AssemblyScript) or in environments where the compilation process is imperceptible to us (ts-node, deno).

WHY DO WE NEED TYPESCRIPT?

- Static typing
- Type Inference
- Documentation and contracts
- Better Tooling (Intelisence / IDE hints)

JAVASCRIPT VS TYPESCRIPT

This is JavaScript code:

```
function test() {
    return 1 + 2;
}
```

This is TypeScript* code:

```
function test() {
    return 1 + 2;
}
```

* - without strict mode enabled

TYPESCRIPT VS JAVASCRIPT

TypeScript is, informally, a superset of JavaScript. Any code written in JS should also be valid code in TS. Thus, all JS elements are available in TypeScript, such as:

- classes,
- symbols,
- generators / iterators,
- Proxy,
- and others.

With TypeScript, you can just use anything there is available in the current version of JavaScript. Also, the developers of TypeScript decided to provide some features that are yet to be in JS, for example:

- decorators,
- class properties,
- operator? to access optional properties,
- and others.

INSTALLATION

Install compiler globally (for current user)

```
npm install -g typescript
```

should be accessible via $PATH env variable, so anywhere:

```
tsc --help
```

TYPESCRIPT COMPILER

Compile to disk:

```
tsc file.ts
tsc file.ts --watch
```

ADDING TYPES

```
function add(a: number, b: number) {
    return a + b;
}
add(1, 2); // Ok
add(1, "2"); // Error!
add("1", "2"); // Error!
add(); // Error!
add([], {}); // Error!
```

STRICT MODE

```
add(null, undefined); // Ok?
```

```
tsc --strict file.ts
```

STRICT MODE

Strict mode enables the following compiler options:

- **noImplicitAny** - Expressions that do not have a type will not become any, they will only cause an error
- **noImplicitThis** - this without a specific type is invalid,instead of becoming any
- **alwaysStrict** - all files are understood in ECMAScript "use strict" mode
- **strictBindCallApply** - the types of the arguments passed to bind, call and apply are more carefully checked
- **strictNullChecks** - stop accepting variables and arguments null and undefined as if they were valid values (like in the previous example)
- **strictFunctionTypes** - function arguments are not bivariate
- **strictPropertyInitialization** - all class fields must be initialized

PLAYGROUND

You can test the code examples and the build result on the square play "prepared by the creators of TypeScipt. You will find it under at:

https://www.typescriptlang.org/play/index.html

EXAMPLE WEB PROJECT

```ts
// project1.ts
const username = window.prompt("What is your name:");
window.alert(`Hello, ${username}`);
```

```html
<!DOCTYPE html>
<html lang="en">
<body>
    <script src="project1.js"></script>
</body>
</html>
```

EXAMPLE NODE PROJECT

```
const user = process.argv[2] || 'Stranger';
console.log(`Hello, ${user}!`);
```

tsc --strict node.ts node node.js

If you are missing types: `npm install -g @types/node`

EXAMPLE FRONTEND - WEBPACK.CONFIG.JS

```
const path = require('path');
module.exports = {
  entry: './src/index.ts',
  output: {
    filename: 'bundle.js',
    path: path.resolve(__dirname, 'dist'),
  },
  module: {
    rules: [
      {
        test: /\.tsx?$/,
        use: 'ts-loader',
        exclude: /node_modules/,
      },
    ],
  },
  resolve: {
    extensions: [ '.tsx', '.ts', '.js' ],
  }
};
```

TOOLS FOR PRODUCTIVITY

Compile and run:

```
npm i -g nodemon
npm i -g ts-node
npm i -g ts-node-dev
```

```
ts-node --compiler-options '{"strict": true}' file.ts
```

Do not use them in production. Check documentation for caveats!

TSCONFIG.JSON

You dont have to remember all compiler arguments. You can use tsconfig.json Generate default config and use `tsc` without extra arguments

```
tsc --init --strict
tsc
```

DEBUGGING NODEJS

```
debugger

node --enable-source-maps --inspect-port=[host:]port index.js
node --enable-source-maps --inspect[=[host:]port] index.js

ts-node-dev --respawn --enable-source-maps --inspect [--inspect-brk] -- ./path/to/app.t
```

JAVASCRIPT REFRESHER + ES6

BUILT-IN OBJECTS

- Number, Nan, Infinity, undefined, null
- eval(), isFinite(), isNaN(), parseFloat(), parseInt(), decodeURL(), encodeURL()
- Object, Function, Boolean, Error(s)
- Number, Math, Date
- String, RegExp
- Array, Object
- JSON
- console

FLOW-CONTROL INSTRUCTIONS

```
if( some_condition == true ){
  // ...
} else {
  // ...
}
```

```
if( some_condition )
operacja ;
```

```
var result = some_condition == true? 'Truthy' : 'Falsy';
```

SWITCH

```
switch( some_expression ){

  case "some value":
    // ...
  break;

  default:
    // ...
}
```

WHILE / DO WHILE LOOP

```
while( some_condition ){
  // Skip to next iteration
  continue;
  // Break out of the loop
  break;
}
```

```
do {
  // Skip to next iteration
  continue;
  // Break out of the loop
  break;

} while( some_condition )
```

FOR LOOPS

```
for(var i; i <= 10; i++) {
  // Skip to next iteration
  continue;
  // Break out of the loop
  break;
}
// i == 10
```

```
var array = [1,2,3];

for( var i in array ) {
  console.log( i, array[i] )
}
// i == 2
```

ITERATING OVER OBJECT KEYS

```
var obj = { a:1, b: 2};

for(var key in obj){

  // Checking if key belongs to object
  // or to one of its prototype parents
  if(obj.hasOwnProperty(key)) {

    console.log(key)

  }
}
```

OPERATORS

```
// Mathematical operators
1 - 2 + 3 * 4 / 5

// Concatenating strings
"ala" + "ma" + "kota"

// Logical operators
true && false || !true

// Binary operators
1<<2  1>>2 ~1

// Comparison operators
== != <= >= === !===

// Type operators
typeof x
x instanceof X
```

COMPARISON OPERATORS

```
// Comparison with typecase
"123" == 123 // true

// Strict Comparison (no typecast)
"123" === 123 // false
```

TYPE CONVERSION

```
parseFloat("123.564") // 123.564

parseInt("123.8") // 123

(123).toString() // "123"

(123).toString(2) // "1111011"

(-2.345).toFixed(2) // "-2.35"

"The answer is " + 42 // "The answer is 42"
"37" + 7 // "377"
"37" - 7 // 30
  + "37" // 37

// Beware of how floating points are stored in memory:
0.1 + 0.2 // 0.30000000000000004
```

FUNCTIONS

Named functions - function declaration

```
function add(a,b){
  return a + b
}
```

Anonymous function - function expression

```
var add = function(a,b){
  return a + b
}
```

Imediately Invoked Function

```
(function(outerParam){
  var innerVariable = "123";

  console.log( outerParam + innerVariable )
})("run with this param")
```

CLOSURES

```
var variable = "value"

function function(parameter){
  return parameter + variable
}

function("from closure")
// "value from closure"
```

VARIABLE SCOPE AND HOISTING

JavaScript variables doesn't jave block scope, but **function scope**:

```
if ( true ) {
  var x = 5;
}
console.log(x); // 5
```

We can use variables that are declared later in code - its so called **"hoisting"**:

```
console.log(y); // exception: y is not defined

console.log(x); // undefined
var x = 5;
```

FUNCTIONAL PROGRAMMING

Build in functional array methods:

```
Array.prototype.every
Array.prototype.some
Array.prototype.forEach
Array.prototype.map
Array.prototype.filter
Array.prototype.reduce
Array.prototype.reduceRight
```

FUNCTION COMBINATORS

Function is first class member - can be passed as variable:

```
function combine( f, g ){
  return function(x){
    return f( g( x ) )
  }
}
```

THIS CONTEXT AND CONTEXT BINDING

```
function WhatIsThis(){
  console.log(this)
}

WhatIsThis() // Window

new WhatIsThis() // Object()

var obj = { test: WhatIsThis, option: 123 }
obj.test() // { test: WhatIsThis, option: 123 }

WhatIsThis.apply({ value: '42' }) // { value: '42' }
```

OBIECT ORIENTED PROGRAMMING

```
function Person(name){

  this.name = name;

  this.sayHello = function(){
    return "Hi, I am " + this.name
  }
}

var alice = new Person('Alice')

alice.sayHello() // "Hi, I am Alice"
```

PROTOTYPAL PROGRAMMING

```
function Person(name){
  this.name = name;
}

Person.prototype.sayHello = function(){
  return "Hi, I am " + this.name
}

var bob = new Person('Bob')
bob.sayHello() // "Hi, I am Bob"
```

PROTOTYPAL INHERITANCE

```
function Employee(name, salary){
  // this.name = name;
  Person.apply(this, arguments)
  this.salary = salary
}
Employee.prototype = Object.create(Person.prototype)

Employee.prototype.doWork = function(){
  return "Done. I would like my "  +this.salary
}

var tom = new Person('Tom',1200)

tom.sayHello() // "Hi, I am Tom"
tom.doWork() // "Done. I would like my 1200"
```

EXCEPTION HANDLING

```
try {
  throw new Error('Upss...');

} catch(err) {
  throw new Error('Failed again ;-( ');

} finally {
  alert('Giving up. Unhandled Error')

}
```

TIMED / ASYNCHRONOUS EVENTS

```
var handler = setTimeout( function(){
  // Call this function "not sooner than" ~16 miliseconds
}, 16)
// handler == 1

// Stop the timeout
clearTimeout(handler)
```

```
var handler = setInterval( function(){
  // Call this function *around* every 1000ms / 1 second
}, 1000)

// Stop interval
clearInterval(handler)
```

WORKING WITH ASYNCHRONOUS CODE

Asynchronous operation - executes as soon as main JavaScript thread is idle

```
setTimeout(()=>{
  console.log('First')

},0)

console.log('Last')

// Last
// First
```

CALLBACK - FUNCTION TO BE CALLED

To get data out of async operation we provide a function as a parameter. Function gets called inside with data once operation completes and data is available.

```
function echo(message, callback){
  setTimeout(function(){
    callback(' ... ' + message)
  }, 1500)
}

echo('Anybody there?', function(response){
  console.log(response)
})

// around 1.5 second later:
// " ... Anybody there?"
```

PROMISES - ASYNCHRONOUS CONTAINERS

Promises allows to avoid nested callback functions. Promise is a value container, which allows to chain operations over asynchronously available data

```
var lazyVal = new Promise(function( resolve, reject ){
  setTimeout(function(){
    resolve("Lazy Value")
  },2000)
})

lazyVal.then(function addBang( value ) {
  return value + "!"
})
.then(function render( valueWithBang ) {
  console.log( valueWithBang)
})
.catch( function handleError( err ){
  console.error( "Promise rejected with " + err )
})
```

SELECTED TOPICS FROM ES6

"ECMASCRIPT 2015" STANDARD - KNOWN AS ES6

Another update to the JavaScript standard after EcmaScript 5. It is available in almost all new browsers except Internet Explorer 11 and a few others.

Current browser support can be checked at: http://kangax.github.io/compat-table/es6/

NEW KINDS OF VARIABLES

```
//block variables - let
if(true){
  let localVar = 1
}
localVar // Exception - local not defined

// stałe - const
const PARAM = 123
const obj = {value:123}

PARAM = "change"
// TypeError: Assignment to constant variable.

obj.value = "ups!" // Written by reference!
```

INTERPOLACJA TEKSTU

```
 // Strings are single-line - break lines
 // requires an escape character before enter

var tekst = " \
";
```

Since ES6 we have multi-line strings with variable interpolation!

```
let variable = 123;
let tekst = ` Title:
  Second line "${ variable + variable }"
`;
// Title:
//    Second line "246"
//
```

OPERATOR SPREAD

```
function f(x, y = 12) {136 / 5000
    // default y value (if y === undefined)
    return x + y;
}
f(3) === 15;

function f(x, ...y) {136 / 5000
  // y is an array of the remaining values
  return x * y.length;
}
f(3, "hello", true) === 6;

function f(x, y, z) {
 return x + y + z;
}
// pass each element of the array separately
f(...[1, 2, 3]) === 6;
```

ARROW FUNCTIONS

```
  const func = (a, b) => {
    return a + b;
  }
```

One parameter allows you to omit the parentheses. The expression in the function body allows you to omit return and braces

```
  const multiply3 = x => x * 3;
  multiply3(2) == 6
```

Lexical `this`

```
var bob = {
  _name: "Bob",
  _friends: [],
  printFriends() {
    this._friends.forEach(f =>
      console.log(this._name + " knows " + f));
  }
};
```

They do not replace **function**. Cannot be used with **new**

DEFAULT FUNCTION PARAMETERS

```
 function f(x, y=12) {
  // y is 12 if not passed (or passed as undefined)
  return x + y;
}
f(3) == 15
```

DESTRUCTURING

```
// list matching
var [a, , b] = [1,2,3];

// swap values
[ b, a ] = [a, b ]

// object matching
var { op: a, lhs: { op: b }, rhs: c }
 = getASTNode() // i.e. { op: 'a',  lhs: {op: 'b' }, rhs: 'c' }}

// object matching shorthand
var {op, lhs, rhs} = getASTNode()

// Can be used in parameter position
function g({name: x}) {
 console.log(x);
}
g({name: 5})
```

MAPS AND SETS

```
// Maps allows you to use any reference as a key
var map = new Map()
map.set( obj, 'secret_value' )

 // The sets check the uniqueness of the added values
var set = new Set()

set.add('x')
// Set(1) {"x"}
set.add('x')
// Set(1) {"x"}
```

SYMBOLS

```
const SPECIAL_TOKEN = Symbol('TOKEN_NAME')

 // Symbol Can Be Used As Key
 map.set (SPECIAL_TOKEN, 'secret')

 // Symbol value is just a description
 map.get ('TOKEN_NAME') == undefined

 // The reference is a unique key,
 // which cannot be "faked
 map.get (SPECIAL_TOKEN) == 'secret'

 // Predefined symbols let you run hidden mechanisms
 // eg Symbol.iterator
```

ITERATORS

```
let fibonacci = {
  [Symbol.iterator]() {
    let pre = 0, cur = 1;
    return {
      next() {
        [pre, cur] = [cur, pre + cur];
        return { done: false, value: cur }
      }
    }
  }
}

for (var n of fibonacci) {
  // truncate the sequence at 1000
  if (n > 1000)
    break;
  console.log(n);
}
```

GENERATORS

```
var fibonacci = {
  [Symbol.iterator]: function*() {
    var pre = 0, cur = 1;
    for (;;) {
      var temp = pre;
      pre = cur;
      cur += temp;
      yield cur;
    }
  }
}

for (var n of fibonacci) {
  // truncate the sequence at 1000
  if (n > 1000)
    break;
  console.log(n);
}
```

CLASSES - "SYNTAX SUGAR" ON PROTOTYPE

```
class Person{

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

  static member = 123

  method(){
    // ...
  }
  field = "value"
}
var pete = new Employee('Pete', 1200 )
pete instanceof Person // true
```

INHERITANCE (PROTOTYPES)

```
class Employee extends Person{

  constructor(name,salary){
    super(name)
  }

  method(){
    super.method()
  }
}

var tom = new Employee('Tom', 1200 )

tom instanceof Employee // true
tom instanceof Person // true
```

MODULES

```
// lib/math.js
export function sum(x, y) {
  return x + y;
}
export var pi = 3.141593;

export default { sum, pi };

// app.js
import * as math from "lib/math";
// default:
// import math from "lib/math";

console.log("2π = " + math.sum(math.pi, math.pi));

// otherApp.js
import {sum, pi} from "lib/math";

console.log("2π = " + sum(pi, pi));
```

TYPESCRIPT - JAVASCRIPT TYPES

Primitives:

- boolean
- number
- string
- symbol
- bigint
- null
- undefined

Complex types:

- object
- Array

TYPESCRIPT - EXTRA TYPES

- tuple,
- enum,
- void,
- any,
- Object,
- never,
- unknown.

BOOLEAN

One of the most basic types. Represents a boolean value, that is, true or false: true, false:

```
const isThisBookOkay: boolean = true;
isThisBookOkay = true
isThisBookOkay = false
```

NUMBER

The numbers in TypeScript are the same as those in JS (TypeScript only adds types).

They are floating point numbers.

There is also the use of hexadecimal, octal and binary literals:

```
const result: number = 123 + 0xbeef + 0b1111 + 0o700;
// 49465
```

STRING

Strings are strings of characters identical to those in JS.

This is probably the most popular data type and without it you wouldn't be able to build any applications.

We pass strings in single (') or double (") quotation marks

```
const firstName: string = "Mateusz";
const lastName: string = "Kulesza";
const hello: string = `Hello,
${firstName} ${lastName}!`;
// Hello,\nMateusz Kulesza!
```

ANY

In some situations, not all type information is available or its declaration would take an inappropriate amount of effort. In these cases, we might want to opt-out of type checking.

```
declare function getValue(key: string): any;

// OK, return value of 'getValue' is not checked
const str: string = getValue("myString");

// Any propagates down to members:
let looselyTyped: any = {};
let d = looselyTyped.a.b.c.d;
//  ^ = let d: any
```

VOID

void is a little like the opposite of any: the absence of having any type at all. You may commonly see this as the return type of functions that do not return a value:

```
function warnUser(): void {
  console.log("This is my warning message");
}
```

Declaring variables of type void is not useful because you can only assign `null` (only if `--strictNullChecks` is not specified) or `undefined` to them

NULL AND UNDEFINED

In TypeScript, both `undefined` and `null` actually have their types named undefined and null respectively. Much like `void`, they're not extremely useful on their own:

```
// Not much else we can assign to these variables!
let u: undefined = undefined;
let n: null = null;
```

By default null and undefined are subtypes of all other types. That means you can assign null and undefined to something like number.

However, when using the `--strictNullChecks` flag, null and undefined are only assignable to unknown, any and their respective types

This helps avoid many common errors. In cases where you want to pass in either a string or null or undefined, you can use the union type `string | null | undefined.`

NEVER

The never type represents the type of values that never occur. For instance, never is the return type for a function that always throws an exception or one that never returns. Variables also acquire the type never when narrowed by any type guards that can never be true.

The never type is a subtype of, and assignable to, every type; however, no type is a subtype of, or assignable to, never (except never itself). Even any isn’t assignable to never.

```
// Function returning never must not have a reachable end point
function error(message: string): never {
  throw new Error(message);
}

// Inferred return type is never
function fail() { return error("Something failed"); }

// Function returning never must not have a reachable end point
function infiniteLoop(): never {  while (true) {} }
```

OBJECT

object is a type that represents the non-primitive type, i.e. anything that is not number, string, boolean, bigint, symbol, null, or undefined.

With object type, APIs like Object.create can be better represented. For example:

```
declare function create(o: object | null): void;
// OK
create({ prop: 0 });
create(null);

create(42);
```

ARRAY

TypeScript, like JavaScript, allows you to work with arrays of values. Array types can be written in one of two ways.

```
// Way 1: Type[]
let list: number[] = [1, 2, 3];Try
The second way uses a generic array type, Array<elemType>:

// Way 2: Array<Type>
let list: Array<number> = [1, 2, 3];

// This is NOT and Array! This is a tuple!
let notAList: [number] = [1]
```

TUPLE

Tuple types allow you to express an array with a fixed number of elements whose types are known, but need not be the same.

```
// Declare a tuple type
let x: [string, number];
// Initialize it
x = ["hello", 10]; // OK


// Initialize it incorrectly
x = [10, "hello"]; // Error
```

More on tuples: https://en.wikipedia.org/wiki/Tuple

TYPE ASSERTIONS

Type assertions are a way to tell the compiler **“trust me, I know what I’m doing.”** A type assertion is like a type cast in other languages, but it performs no special checking or restructuring of data. It has no runtime impact and is used purely by the compiler. TypeScript assumes that you, the programmer, have performed any special checks that you need.

```
// the as-syntax:
let someValue: unknown = "this is a string";

let strLength: number = (someValue as string).length;Try

//  “angle-bracket” syntax:
let someValue: unknown = "this is a string";

let strLength: number = (<string>someValue).length;
```

ABOUT NUMBER, STRING, BOOLEAN, SYMBOL AND OBJECT

It can be tempting to think that the types `Number`, `String`, `Boolean`, `Symbol`, or `Object` are the same as the lowercase versions recommended above. These types do not refer to the language primitives however, and almost never should be used as a type.

```
const s:String = 'text' // Wrong

const s:string = 'text' // Correct!
```

COMPLEX TYPES - INLINE

Complex object types can be defined inline

```
const item: { name:string, value:number } = {name:'item', value: 42 }

function printItem(item:  { name:string, value:number }) {
    return item.name + ' : ' + item.value
}

// Or can be aliased for reuse
type Item = { name:string, value:number };
```

DECLARING COMPLEX TYPES - TYPE ALIAS VS INTERFACE

Unlike an `interface` declaration, which always introduces a named object type, a `type` alias declaration can introduce a name for any kind of type, including primitive, union, and intersection types.

```
type aString = string

type x = string | number
type y = { field: string }

interface Z { field: string }
```

Interfaces can also be used for merging declarations. On that later...

NAMED INTERFACES

Simple interface

```
interface User{
    id: number
    name: string
    active: boolean
}
```

OPTIONAL PROPERTIES

Not all properties of an interface may be required. Some exist under certain conditions or may not be there at all. These optional properties are popular when creating patterns like “option bags” where you pass an object to a function that only has a couple of properties filled in.

```
interface SquareConfig {
  color?: string;
  width?: number;
}

const square: SquareConfig = {}
square.color // undefined
square.color.toUpperCase() // Error!

if(square.color){
    quare.color.toUpperCase() // OK
}
```

OPTIONAL PROPERITIES AND OPTIONAL CHAINING

At its core, optional chaining lets us write code where TypeScript can immediately stop running some expressions if we run into a `null` or `undefined`.

```
// insead of
let x = foo && foo.bar && foo.bar.baz()

// you can write
let x = foo?.bar?.baz();

// Actually it only checks for undefined or null. Not 'falsy' values like "0" or ""
let x = foo === null || foo === undefined ? undefined : foo.bar.baz();
```

NON-NULL ASSERION OPERATOR

A new `!` post-fix expression operator may be used to assert that its operand is non-null and non-undefined in contexts where the type checker is unable to conclude that fact.

```
let x = foo!.bar!.baz();
```

Operation `x!` produces a value of the type of `x` with `null` and `undefined` excluded.

Similar to the forms `<T>x` and `x as T`, the `!` non-null assertion operator is simply removed in the emitted JavaScript code.

NULLISH COALESCING

You can think of this feature - the ?? operator - as a way to “fall back” to a default value when dealing with null or undefined. When we write code like

```
let x = foo ?? bar();

// instead of:
let x = foo !== null && foo !== undefined ? foo : bar();
```

READONLY PROPERTIES

Some properties should only be modifiable when an object is first created.

```
interface Point {
  readonly x: number;
  readonly y: number;
}
// After the assignment, x and y can’t be changed.
let p1: Point = { x: 10, y: 20 };
p1.x = 5; // error!
// Error: Cannot assign to 'x' because it is a read-only property.
```

Readonly is for properties only. Variables use const whereas properties use readonly.

INDEXABLE TYPES

Similarly to how we can use interfaces to describe function types, we can also describe types that we can “index into” like `a[10]`, or `ageMap["daniel"]`. Indexable types have an index signature that describes the types we can use to index into the object, along with the corresponding return types when indexing.

```
interface StringArray {
  [index: number]: string;
}

let myArray: StringArray;
myArray = ["Bob", "Fred"];

let myStr: string = myArray[0];
```

EXTRACTING TYPES

```
type UserID = User["id"];
function findUser(id: User["id"]): User { /* … */ }
```

DECLARATION MERGING

Interfaces with same name are merged allowing for extending existing types.

```
interface Xyz {
  a: number;
}

interface Xyz {
  b: string;
}

const res: Xyz = {
  a: 1,
  b: "foo",
};
```

FUNCTIONS

```
// Named function
function add(x, y) {
return x + y;
}

// Anonymous function
let myAdd = function (x, y) {
return x + y;
};
```

TYPING THE FUNCTION

```
function add(x: number, y: number): number {
  return x + y;
}

let myAdd = function (x: number, y: number): number {
  return x + y;
};
```

WRITING THE FUNCTION TYPE

We can write complete function signature 'outside'

```
let myAdd: (baseValue: number, increment: number) => number = function (
  x: number,
  y: number
): number {
  return x + y;
};
```

FUNCTION SIGNATURE AS TYPE

When function has its type defined then arguments and return type can be inferred:

```
let myAdd2: (baseValue: number, increment: number) => number = function (x, y) {
  return x + y;
};
```

FUNCTION ALIAS AND INTERFACE

We can extract function type altogether

```
type addFunc: (baseValue: number, increment: number) => number;
let myAdd:addFunc = (x, y) => x + y;

interface SearchFunc {
  (source: string, subString: string): boolean;
  (source: string, subString: string, startPos:number): boolean; // overload
}
const mySearch: SearchFunc = (heap, needle, start) => heap.substring(needle, start);
```

OPTIONAL AND DEFAULT PARAMETERS

```
function buildName(firstName: string, lastName: string) {}

let result1 = buildName("Bob"); // error, too few parameters
let result2 = buildName("Bob", "Adams", "Sr."); // error, too many parameters
```

We can make params optional or give them defaults

```
function buildName(firstName: string = 'Guest', lastName?: string) {}

let result1 = buildName(); // OK, firstname == 'Guest'
let result2 = buildName("Bob"); // OK, lastName == undefined
```

REST PARAMETERS

```
function sum(initialValue = 0, ...rest : number[]): number {
    return rest.reduce( ( sum, x ) => sum + x , initialValue)
}
```

FUNCTION THIS CONTEXT

```
interface Elem{
    onclick: (this:Elem, event:Event) => void
}
```

ENUMS

```
enum InvoiceStatus {
  SUBMITTED, // 0
  APPROVED, // 1
  PAID, // 2
}

function getStatusLabel(status: InvoiceStatus) {
  switch (status) {
    case InvoiceStatus.SUBMITTED:
      return "invoice was submitted";
    case InvoiceStatus.APPROVED:
      return `it's approved`;
    case InvoiceStatus.PAID:
      return "invoice paid";
  }
}

// Two-way mapping
const val = InvoiceStatus.SUBMITTED; // 0
const statusName = InvoiceStatus[val]; // 'SUBMITTED
```

INITIALIZE ENUMS

```
enum AnotherEnum {
  SUBMITTED = 123,
  APPROVED, // 124
  PAID, // 125
  SOMETHING_ELSE = 1000 + 24, // expression
  MORE, // 1025
}

enum FileAccess {
  None = 1 << 0,
  Read = 1 << 1,
  Write = 1 << 2,
  Execute = 1 << 3,
  ReadWrite = FileAccess.READ | FileAccess.WRITE;
}
```

STRING ENUMS

```
const UserRole = {
  USER: "user",
  ADMIN: "admin",
  MODERATOR: "moderator",
};

function getPermissionsFor(
  role: Role,
): number | undefined {
  switch (role) {
    case Role.ADMIN:
      return 123;
    case Role.USER:
      return undefined;
    default:
      const _never:never = role; // exhaustiveness check
      return _never
  }
}
```

FUNCTION LITERAL OVERLOADS

```
function create(name: "User"): User;
function create(name: "Admin"): Admin;
function create(name: "Moderator"): Moderator;
function create(name: string): Character {
// …
}
const m = create("Moderator"); // m is a Moderator
```

TYPE INTERSECTIONS AND UNIONS

```
type A = {
  a: string;
  b: number;
};
type B = {
  b: number;
  c: string;
};

type Union = A | B;
type Intersection = A & B;
```

RECURSIVE TYPES

```
type User = {
  name: string;
  friends: Array<User>;
};

type Json =
| null
| string
| number
| boolean
| Json[]
| { [name: string]: Json };
```

LITERAL TYPES

A literal is a more concrete sub-type of a collective type. What this means is that "Hello World" is a string, but a string is not "Hello World" inside the type system.

```
const anyText:string = 'Hello!'
anyText = 'Goodbye!' // OK

const helloText: 'Hello' = 'Hello'
helloText = 'Goodbye!' // Error!

// alternatively:
// const helloText = 'Hello' as const
```

LITERAL UNIONS

You can alias union of multiple possible literals

```
type Easing = "ease-in" | "ease-out" | "ease-in-out";

class UIElement {
  animate(dx: number, dy: number, easing: Easing) {
    if (easing === "ease-in") {
    } else if (easing === "ease-out") {
    } else if (easing === "ease-in-out") {
    } else {
      // It's possible that someone could reach this
      // by ignoring your types though.
    }
  }
}
let button = new UIElement();
button.animate(0, 0, "ease-in");
button.animate(0, 0, "uneasy"); // Error!
```

UNION ALTERNATIVE SYNTAX

```
type RequestCredentials =
  | "omit"
  | "same-origin"
  | "include";

let cred: RequestCredentials;
cred = "omit"; // OK
cred = "inclade"; // Error!
```

INDEXED TYPES AND UNIONS

```
type Headers = "Accept" | "Authorization" /* … */;
type RequestConfig = {
  [header in Headers]: string;
};

const requestConfig: RequestConfig = {
  Accept: "…",
  Authorization: "…",
};
```

CLASSES

```
class Person {
  name:string

  constructor(){}

  method(){}
}
```

CLASS UNINITIALIZED PROPERTIES

```
class SomeClass{
  initializedProperty = 'some default text';

  uninitializedProperty:string // Error in strict mode!

  optionalProperty?:string // undefined when instance is created

  // You can make compiler accept it but its not safe
  trustMeIKnowWhatIAmDoingProperty!:string

  constructor(){
    // You have to remember to initialize it yourself
    this.trustMeIKnowWhatIAmDoingProperty = 'its initialized now'
  }
}
```

CLASS PROPERTIES ACCESS MODIFIERS

```
class IAmPartiallyAccesible{
  // fields are publicly accesible by default
  /* public  */ name = ''

  // only accessible in instances of this class
  private _secretData = ''

  // accessible in instances of this class and sub-classes
  protected _overrideMe = ''

  // Cannot be changed after instance is created
  readonly _cantChangeMe = ''

}

class AnotherClass extends IAmPartiallyAccesible{
  method(){
    this._secretData // Error
    this.name // OK
    this._overrideMe // OK
  }
}
```

INITIALIZE FIELDS WITH ARGUMENTS

```
class ExampleDataDTO{
  public name: string = ''; private timestamp: string = '';

  constuctor(name:string, timestamp:string){
    this.name = name; this.timestamp = timestamp
  }
}
```

Can be written as

```
class SimplerDataDto{
  constructor(public name:string, private timestam:string){}

  toJSON(){
    return { name: this.name } // fields are populated from arguments!
  }
}
```

Any modifier works: public, private, protected, readonly.

CLASS INHERITANCE

```
class Employee extends Person {
  name:string

  constructor(){
    // parent constructor call is required!
    super()
  }

  method(){
    // optionally can call overriden methods
    super.method()
  }
}
```

CLASS CAN IMPLEMENT INTERFACE OR TYPE

```
interface IDoStuff {
    // optionally can define constructor signature:
    new (name:string): IDoStuff
    name: string
    method():void
}
type IHaveId = {id:string}

// you can implement multiple types
class StuffClass implments IDoStuff, IHaveId{
  name: string
  constructor(){}
  method(){}
}
```

GENERICS

The key motivation for generics is to document meaningful type dependencies between members. The members can be:

- Class instance members
- Class methods
- function arguments
- function return value

```
const x: Array<string> = ["a", "b", "c"];
```

GENERIC TYPE EXAMPLE

```
type Ref<T> = {
  current: T;
};
const ref1: Ref<number> = { current: 123 };
const ref2: Ref<string> = { current: "aaa" };

function getValue<T>(ref: Ref<T>): T {
  return ref.current;
}
```

GENERIC CLASS EXAMPLE

```
/** A class definition with a generic parameter */
class Queue<T> {
  private data = [];
  push(item: T) { this.data.push(item); }
  pop(): T | undefined { return this.data.shift(); }
}
```

GENERICS AND CONSTRAINTS

Those use generics but do not offer any type safety:

```
function parse<T>(name: string): T {}
function serialize<T>(name: T): string {}
```

practivally same as:

```
declare function parse(name: string): any;
const something = parse('something') as TypeOfSomething;
```

GENERICS AND CONSTRAINTS

Use generic type in two places to add constraints:

```
const id = <T>(x: T): T => x;
const result = id<number>(1);

// typeof result == number
```

Here generics provide constraints between argument and return value

GENERIC INTERFACE

new keyword declares constructor signature - object can be constructed with new

```
interface Constructable<T> {
  new (...args: any[]): T;
}
```

GENERICS AND TYPE CONSTRAINTS - EXTENDS

Generic works like unknown type until type is given or inferred. Type can be constraint to only subtypes

```
type ObjWithName = { name: string };

function printName<T extends ObjWithName>(arg: T) {}

printName({ name: "Kate" }); // OK
printName({ name: "Michael", age: 22 }); //  OK
printName({ age: 22 }); // Error!
```

MULTI-TYPE GENERICS

```
function makePair<T, U>(arg1: T, arg2: U): [T, U] {
  return [arg1, arg2];
}

function defaults<T extends object, U extends T>(
  obj1: T,obj2: U,
) {
  return { ...obj1, ...obj2 };
}
defaults({a:0, b:1}, { a: 123  });
// OK, { a: 123, b:1 }
defaults({a:0, b:1}, { a: 123, c: 234 });
// Error U contains 'c' witch T does not!
```

TYPE COMPABILITY

```
let str: string = "Hello";
let num: number = 123;

str = num; // ERROR: `number` is not assignable to `string`
num = str; // ERROR: `string` is not assignable to `number`
```

STRUCTURAL TYPING / DUCK-TYPING

```
interface Point {
    x: number,
    y: number
}

class Point2D {
    constructor(public x:number, public y:number){}
}
let p: Point;
// OK, because of structural typing
p = new Point2D(1,2);
```

STRUCTURAL OVERLOADS

```
interface Point2D {
    x: number; y: number;
}
interface Point3D {
    x: number; y: number; z: number;
}
var point2D: Point2D = { x: 0, y: 10 }
var point3D: Point3D = { x: 0, y: 10, z: 20 }
function iTakePoint2D(point: Point2D) { /* do something */ }

iTakePoint2D(point2D); // exact match okay
iTakePoint2D(point3D); // extra information okay
iTakePoint2D({ x: 0 }); // Error: missing information `y`
```

VARIANCE

Variance is an easy to understand and important concept for type compatibility analysis.

For simple types Base and Child, if Child is a child of Base, then instances of Child can be assigned to a variable of type Base. -- This is `polymorphism 101`

Compability can take multiple "flavours":

- Covariant : (co aka joint) only in same direction
- Contravariant : (contra aka negative) only in opposite direction
- Bivariant : (bi aka both) both co and contra.
- Invariant : if the types aren't exactly the same then they are incompatible.

UNIONS

You can make type from union of types

```
type Id = number | string

interface Square {
    kind: "square"; size: number;
}

interface Rectangle {
    kind: "rectangle"; width: number; height: number;
}
type Shape = Square | Rectangle;
```

TYPE GUARDS

```
function parseInt(arg: string | number) {
  if (typeof arg === "string") {
    arg.toUpperCase()
  } else {
    arg.toPrecision(2)
  }
}
```

TYPE GUARDS - INSTANCEOF

```
class Character { name!: string;  }
class User extends Character { age!: number; }
class Enemy extends Character { hp!: number; }

declare let ch: Character;

if (ch instanceof User) {
  ch.age; // OK
  ch.name; // OK
} else if (ch instanceof Enemy) {
  ch.hp; // OK
  ch.name; // OK
} else {
  ch.name; // OK
}
```

TYPE GUARDS - IN OPERATOR

```
type Character = { name: string; };
type User = { name: string; age: number; };
type Admin = { name: string; age: number; role: string; };
declare let x: Character | User | Admin;

if ("age" in x) {
  x; // User | Admin
}
if ("role" in x) {
  x; // Admin
}
```

EXTRACTING TYPES

```
const defaultConfig = {
  port: 3000,
  host: "localhost",
};
type Config = typeof defaultConfig;
```

REUSABLE TYPE GUARDS

```
type SingleValue = { value: string };
type ManyValues = { options: Array<SingleValue> };

function isSingleValue(val: SingleValue | ManyValues): val is SingleValue {
  return ( "value" in val && typeof val.value === "string" );
}
declare const obj: any;

if (isSingleValue(obj)) {
  // obj is a SingleValue
} else {
  // any
}
```

DISCRIMINATED UNIONS

If all members of union share a property but each have unique literal value, we can discrimiate:

```
function area(s: Shape) {
    if (s.kind === "square") {
        // Now TypeScript *knows* that `s` must be a square ;)
        // So you can use its members safely :)
        return s.size * s.size;
    }
    else {
        // Wasn't a square? So TypeScript will figure out that it must be a Rectangle
        // So you can use its members safely :)
        return s.width * s.height;
    }
}
```

EXHAUSTIVENESS CHECK

```
type Shape = Square | Rectangle | Circle // !!!

function area(s: Shape) {
    if (s.kind === "square") {
        return s.size * s.size;
    }
    else if (s.kind === "rectangle") {
        return s.width * s.height;
    }
    else {
        // We can check for that with 'never':
        const _exhaustiveCheck: never = s;
        // ERROR : `Circle` is not assignable to `never`
    }
}
```

TYPE MAPPING

Type alias allows you to extract or derive new types from existing ones

```
interface Payload{ id:string, name:string, extra:string }

type someKeys = 'id' | 'name'
type PartialPayload = {
  // new Type borrows only few keys and types from original type
  [key in someKeys]: Payload[key]
}
```

TYPE MAPPING WITH GENERICS

Using generics makes mapping reusable

```
interface Payload{ id:string, name:string, extra:string }

type Partial<T> = {
  // take all keys and all types, but make them optional! (?)
  [key in keyof T] ? : T[key]
}
const changedFields: Partial<Payload> = { name: 'I am enough!' }
```

You dont have to define `Partial` yourself. Its built-in typescript type!

CONDITIONAL TYPES

Type can differ depending on generic parameter types

```
type R = T extends U ? X : Y;

// Simple usage example:
type IsBoolean<T> = T extends boolean ? true : false;
type t01 = IsBoolean<number>; // false
type t02 = IsBoolean<string>; // false
type t03 = IsBoolean<true>; // true
```

CONDITIONAL TYPES AND UNIONS

```
type NonNullable<T> = T extends null | undefined
  ? never
  : T;

type t04 = NonNullable<number>; // number
type t05 = NonNullable<string | null>; // string
type t06 = NonNullable<null | undefined>; // never


type StringsOnly<T> = T extends string ? T : never;
type Result = StringsOnly<"abc" | 123 | "ghi">;
// "abc" | never | "ghi", therefore only "abc" | "ghi"
```

TESTING WITH MOCHA, CHAI AND TYPESCRIPT

https://mochajs.org/

```
npm i -D chai @types/chai chai-as-promised @types/chai-as-promised
         sinon @types/sinon sinon-chai
```

MOCHA, CHAI AND SINON

Mocha works best with assertions lib like chai and mocking lib like sinon

```
// testSetup.ts
import chai from "chai";
import chaiAsPromised from "chai-as-promised";
import sinonChai from "sinon-chai"

declare global { var expect: Chai.ExpectStatic }
global.expect = chai.expect

chai.use(chaiAsPromised);
chai.use(sinonChai);
```

WRITING TESTS IN MOCHA AND CHAI

```
// src/some.test.ts
describe('Some module under test', () => {
  it('does X',() => {
    expect('X').to.eq('X')
  })
})
```

Run in watch mode:

```
mocha -r ts-node/register -r src/testSetup.ts --extension ts --watch src/**/*.test.ts
```

TESTING PROMISES WITH CHAI-AS-PROMISED

```
it('Test Promise', async () => {
    const p = Promise.resolve(123)
    await expect(p).eventually.eq(123)

    const p = Promise.reject(123)
    await expect(p).eventuallyrejectedWith(123)
});
```

SPYING FUNCTION CALLS WITH SINON

```
it('Test Mock', () => {
  const spy = sinon.spy()

  function hello(name: string, cb: (t: string) => void) { cb('hello ' + name) }
  hello('foo', spy)

  expect(spy).to.have.been.calledWith('hello foo')
})
```

MOCKING INSTANCES

```
class Test { someMethod(x: string) { return x } }

it('Mocks Test',() => {
      // Arrange
      const stub = sinon.createStubInstance(Testing)
      stub.someMethod.returns('test')

      // Act
      stub.someMethod('test')

      // Assert
      expect(stub.someMethod).calledWith('test').and.returned('test')
})
```

MOCKING MODULE DEPENDENCIES

```
import postsApi from './postsApi'
import postsService from './postsService'

it('Mock dep', () => {
  const fetchPostsStub = sinon.stub(postsApi,'fetchPost')

  postService.loadPost(123)

  expect(fetchPostsStub).to.have.been.calledWith(123)
})
```

EXPRESS.JS

```
npm i express @types/express
```

```
import express from "express";
var app = express();

app.get("/", function(req, res) {
  res.status(200).send("<h1>GET request to homepage</h1>");
});

const HOST = '0.0.0.0'; const PORT = 3000;

app.listen(HOST,PORT, () => { console.log(`⚡[server]: Listening on http://${HOST}:${PO
```

REQUEST PARAMETERS

```
app.get("/user/:id", function(request, response) {
  response.send("user " + request.params.id); // /user/123
});
```

```
app.get("/user/", function(request, response) {
  response.send("user " + request.query.query_param); // ?query_param=123
});
```

ROUTER TYPES

```
app.post<
  ParamsDictionary = {},
  ResponseBody = {}
  RequestBody = {}
  RequestQuery = {}
>("/user/", function(request, response) { /* ... */ });
```

MIDDLEWARE

```
app.use((req, res, next) => {
  // Handle request
  // ...
  // Handle error
  next(err);
  // or forward req to next middleware:
  next();
  // or go back to routing
  next('route')
});
```

ROUTING

```
const routes = express.Router()
  .use(/* router middleware */)
  .get("/", (req, res) => {
    /*  */
  })
  .post("/", (req, res) => {
    /*  */
  });

app.use("/mount/sub-routes/here", routes);
```

ERROR HANDLING

```
// always use an default express/connect error handling middleware
// it will be called if any errors occur in the domain
// see http://expressjs.com/en/guide/error-handling.html
app.use(function(err, req, res, next) {
  res.status(err.status || 500);
});
```

BUILTIN MIDDLEWARES

- **express.static** serves static assets such as HTML files, images, and so on.
- **express.json** parses incoming requests with JSON payloads.
- **express.urlencoded** parses incoming requests with URL-encoded payloads

3RD PARTY MIDDLEWARES

- **cors** Enable cross-origin resource sharing (CORS) with various options.
- **helmet** Protect from various exploits.
- **errorhandler** Development error-handling/debugging.
- **morgan** HTTP request logger.
- **multer** Handle multi-part form data.
- **session** Establish server-based sessions (development only).

EXAMPLE SETUP

```
app.use(express.static(path.join(__dirname, "public")));
app.use(express.static(path.join(__dirname, "files")));

// parse application/x-www-form-urlencoded
app.use(bodyParser.urlencoded({ extended: false }));
// parse application/json
app.use(bodyParser.json());
```

SESSIONS

```
npm i express-session @types/express-session
```

```
var app = express();
app.set("trust proxy", 1); // trust first proxy
app.use(
  session({
    secret: "keyboard cat",
    resave: false,
    saveUninitialized: true,
    cookie: { secure: true, maxAge: 60000 }
  })
);
```

https://github.com/expressjs/session

REQUEST - SESSION OBJECT

```
app.get('/', function(req, res, next) {
  if (req.session.views) {
    req.session.views++
    res.send('Welcome back! Its your'+req.session.views+' visit!')
  }else{
    req.session.views = 1
    res.send('Hello! Its your first time with us')
  }
```

EXTENDING BUILT-IN TYPES

```
declare global {
    namespace Express {
        // Inject additional properties on express.Request
        interface Request { }
    }
}
// or extend session interface:

declare module 'express-session' {
    interface SessionData {
        views: number;
    }
}
```

declaration merging

https://www.typescriptlang.org/docs/handbook/declaration-merging.html

FILE UPLOAD

```
<form action="/profile" method="post" enctype="multipart/form-data">
  <input type="file" name="avatar" />
</form>
```

```
var multer = require("multer");
var upload = multer({ dest: "uploads/" });
var cpUpload = upload.fields([
  { name: "avatar", maxCount: 1 },
  { name: "gallery", maxCount: 8 }
]);

app.post("/cool-profile", cpUpload, function(req, res, next) {
  // req.files is an object (String -> Array)
  // where fieldname is the key, and the value is array of files
  fs.rename(req.files["avatar"][0].path, destination_path, err => {
    /* file uploaded and moved to `destination_path` */
  });
});
```

REST

```
app.get("/users", (req, res) => {
  return res.send("GET method on user resource");
});

app.post("/users", (req, res) => {
  return res.send("POST method on user resource");
});

app.put("/users/:userId", (req, res) => {
  return res.send(`PUT method on user/${req.params.userId} resource`);
});

app.delete("/users/:userId", (req, res) => {
  return res.send(`DELETE method on user/${req.params.userId} resource`);
});
```

SUPERTEST

```
var app = require("../");
var supertest = require("supertest")(app);

it("Responds with 'Hello, World!'", function(done) {
    supertest
        .get("/")
        .expect(200)
        .expect({
            greeting: "Hello, World!"
        })
        .end(done);
});
```

DECORATORS

```
{
  "compilerOptions": {
    "target": "ES5",
    "emitDecoratorMetadata": true,
    "experimentalDecorators": true
  }
}
```

https://www.typescriptlang.org/docs/handbook/decorators.html

USING DECORATORS - EXAMPLE

```
import { JsonController, Param, Body, Get, Post, Put, Delete } from 'routing-controller

@JsonController('/users')
export class UserController {

  @Get('/')
  getAll(@Req() request: any, @Res() response: any) {
    return []
  }
}
useExpressServer(app, {
  routePrefix: '/api',
  controllers: [UserController], // we specify controllers we want to use
});
```

DECORATORS - VALIDATION

https://github.com/typestack/class-validator

```
enum Roles {
    Admin = "admin",
    User = "user",
    Guest = "guest",
}

class GetUsersQuery {

    @IsPositive()
    limit: number;

    @IsAlpha()
    city: string;

    @IsEnum(Roles)
    role: Roles;

    @IsBoolean()
    isActive: boolean;

}
```

DECORATORS - PARSING

https://github.com/typestack/class-transformer

```
export class User {
  firstName: string;
  lastName: string;

  @Exclude({ toPlainOnly: true }) password: string;

  @Type(() => Album) albums: Album[];

  getName(): string {
    return this.lastName + ' ' + this.firstName;
  }
}

@Controller()
export class UserController {
  post(@Body() user: User) {
    console.log('saving user ' + user.getName());
  }
}
```

DECORATORS - TYPEORM ENTITY

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

@Entity()
export class Student {

    @PrimaryGeneratedColumn()
    id: number;

    @Column()
    Name: string;

    @Column()
    age: number;
}
```

Notes:

TYPEORM

```
npm install typeorm -g
npm install typeorm reflect-metadata @types/node

package.json:
```json
"scripts":{
  "typeorm": "node --require ts-node/register ./node_modules/typeorm/cli.js",
```

npm install mysql // mysql npm install pg // postgresql npm install sqlite3@5.0.0 // sqlite3 ```
```

TYPEORM CLI

https://github.com/typeorm/typeorm/blob/master/docs/using-cli.md

typeorm init --name FirstProject --database mysql

```
  "typeorm": "node -r ts-node/register ./node_modules/typeorm/cli.js",
  "db": "docker run --rm --name db -p 5432:5432 -e POSTGRES_PASSWORD=mysecretpassword -
```

RELATIONSHIPS

- **one-to-one** − One object of the given entity relates to only one object of the target entity and vice versa. For example, a country will have only one capita city and similarly a city will be capital of only one country.
- **many-to-one** − Multiple object of the given entity relates to one object of the target entity. For example, city comes under only one country but country can have multiple cities.
- **one-to-many** − Same as many-to-one except the relationship is reversed.
- **many-to-many** − Multiple object of the given entity relates to multiple objec of the target entity. For example, an article may be tagged under multiple topic like programming language, finance, etc., and at the same time a particular tag may have multiple articles as well.

AUTOLOADING

TypeORM also provides options to enhance the relationship of the entities.

They are as follows −

- **eager** − Source entity object loads the target entity objects as well.
- **cascade** − Target entity object gets inserted or updated while the source entity object is inserted or updated.
- **onDelete** − Target entity objects get deleted as well while the source entity object is deleted.
- **primary** − Used to specify that the relation column is primary key or not.
- **nullable** − Used to specify that the relation column is nullable or not.

DEPLOYMENT

https://pm2.keymetrics.io

```
npm i pm2 -g

pm2 start app.js --watch
pm2 start app.js -i 10

pm2 [list|ls|status]

pm2 restart app_name
pm2 reload app_name
pm2 stop app_name
pm2 delete app_name

pm2 restart all

pm2 logs --lines 200
pm2 monit
```

https://pm2.keymetrics.io/docs/usage/quick-start/#cheatsheet

START MULTIPLE APPLICATIONS

```
// cosystem.config.js
module.exports = {
  apps: [
    {
      name: "app",
      script: "./app.js",
      env: {
        NODE_ENV: "development"
      },
      env_production: {
        NODE_ENV: "production"
      }
    },
    {
      name: "worker",
      script: "worker.js"
    }
  ]
};
```

```
pm2 ecosystem
```