




# TYPESCRIPT

Parce-que vanilla JavaScript est un rien... limité ?

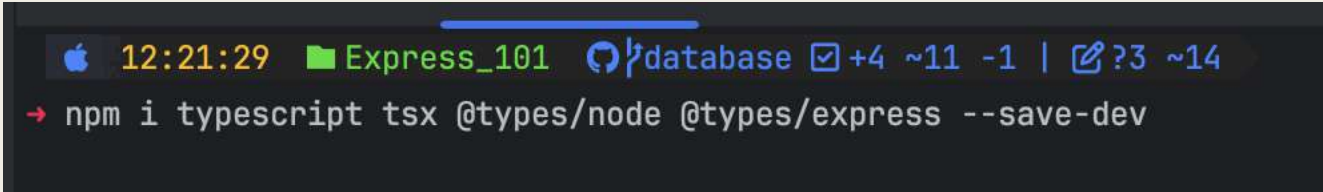


# Pourquoi ?

- On vient de construire notre toute notre application en JavaScript et elle marche alors pourquoi rajouter de la complexité avec TypeScript ?
- Passer à TypeScript va nous permettre de consolider notre code avec des informations de types, ce qui va nous permettre de plus facilement déceler les bugs dans une application grandissante.
- De plus, TS est un Superset de JS, donc du code JavaScript valide reste du code TypeScript valide, ce qui va nous permettre de migrer notre code avec assez peu de complexité

# Installer les dépendances

- Pour que notre application fonctionne, il va nous falloir quelques dépendances :
  - *Typescript, le transpileur lui-même*
  - *Tsx, TypeScript Execute pour lancer notre application*
  - *Des dossiers de types*



```
12:21:29 Express_101 database +4 ~11 -1 | ?3 ~14  
→ npm i typescript tsx @types/node @types/express --save-dev
```

# Modifier notre package.json

- Pour lancer notre application, nous n'allons donc plus utiliser Nodemon mais « tsx watch »

```
package.json x
1  {
2    "name": "app",
3    "version": "1.0.0",
4    "type": "module",
5    "description": "",
6    "main": "server.js",
7    "scripts": {
8      "dev": "tsx watch server.js"
9    },
10   "author": "Jean-François DI RIENZO",
11   "license": "ISC",
12   > "dependencies": {"dotenv": "^16.4.5"...},
13   > "nodemonConfig": {"ext": "js,mjs,cjs,json,twig,html"...},
14   > "devDependencies": {"@types/express": "^5.0.0"...}
15 }
16
17
18
19
20
21
22
23
24
25
26
27
28
29
30
```

```

tsconfig.json
1  {
2      "compilerOptions": {
3          "target": "esnext",
4          "module": "esnext",
5          "outDir": "./dist",
6          "strict": true,
7          "esModuleInterop": true,
8          "skipLibCheck": true,
9          "forceConsistentCasingInFileNames": true,
10         "moduleResolution": "node",
11         "baseUrl": ".",
12         "paths": {
13             "*": [
14                 "node_modules/*",
15                 "*"
16             ]
17         },
18     },
19     "include": [
20         "*"
21     ],
22     "exclude": [
23         "node_modules"
24     ]
25 }

```

# tsconfig.json

- C'est la partie la plus compliquée de notre configuration, il va falloir spécifier comme nous voulons que le transpiler TS se comporte.
- Il faut bien comprendre que TypeScript n'est pas réellement un langage de programmation mais plutôt un « méta langage » qui va finir compilé en du code JavaScript pour être interprété ensuite.
- JavaScript ayant des versions différentes, il va falloir spécifier quelle sera la « cible » de notre transpilation (vers quelle version de JavaScript sera transformé notre code)

# Définir nos types

- La première chose que nous allons faire est de définir nos types, le but est d'avoir tout de suite une image plus générale de ce que nous allons manipuler dans notre application
- Commençons par le store

```
TS store.d.ts ×  
1  import {Connection} from 'mysql2/promise'  
2  import {Todo} from './todo';  
3  
   Show usages new *  
4  export interface Store {  
5      readonly todoStore: TodoStore  
6  }  
7  
   Show usages new *  
8  export interface TodoStore {  
9      readonly database: Connection  
10     getAll: () => Promise<Todo[]>  
11     getById: (id: number) => Promise<Todo>  
12     add: (todo: Todo) => Promise<Todo>  
13     update: (todo: Todo) => Promise<Todo>  
14     delete: (id: number) => Promise<void>  
15 }  
16
```

# Ce que ça va donner dans notre objet

- On va pouvoir déclarer que notre classe implémente cette interface et s'assurer que nos types sont bien respectés

```
TS TodoStore.ts x
1 import {Connection} from "mysql2/promise";
2 import type {TodoStore as TodoStoreInterface} from "../types/store";
3 import {Todo} from "../types/todo";
4
5 Show usages John-Bob-DIRIENZO *
6 export class TodoStore implements TodoStoreInterface {
7
8   Show usages John-Bob-DIRIENZO *
9   constructor(database: Connection) {
10     this.database = database
11   }
12
13   no usages John-Bob-DIRIENZO *
14   async getAll(): Promise<Todo[]> {
15
16   no usages John-Bob-DIRIENZO *
17   async getById(id: number): Promise<Todo> {
18
19   Show usages John-Bob-DIRIENZO *
20   async add(todo: Todo): Promise<Todo> {
21
22   no usages new *
23   async update(todo: Todo): Promise<Todo> {
24
25   Show usages new *
26   async delete(id: number): Promise<void> {
27
28   }
```

# Simplifier des déclarations

```
Store.ts x
1 import {TodoStore} from "../TodoStore.js";
2 import {Connection} from "mysql2/promise";
3 import {Store} from "../types/store";
4
5 Show usages John-Bob-DIRIENZO *
6 export function NewStore(database: Connection): Store {
7     return {
8         todoStore: new TodoStore(database)
9     }
10 }
```

On va aussi pouvoir simplifier la déclaration de notre constructeur du Store parent en le mettant dans une simple fonction



# Simplifier des déclarations

- On va pouvoir simplifier encore plus notre déclaration pour le contexte puisque ce n'est qu'un ensemble d'éléments sans réelle logique métier

```
TS context.d.ts x
1 import {Store} from "../store";
2
3 Show usages new *
4 export interface Context {
5     store: Store
6 }
```

```
TS server.ts x
1 > import ...
9
10 dotenv.config()
11
12 const app : Express = express()
13 const port = 3001
14
15 app.use(express.json())
16 app.use(express.urlencoded({extended: true}))
17
18 app.use(logger)
19
20 > const database : Connection = await createConnection({host: process.env.DB_HOST...})
27
28 const store : Store = NewStore(database)
29 const ctx: Context = {
30     store: store
31 }
```

