TechReturners

License:

Mars Rover Additional Info - TypeScript

1 What's In This Document?

Part of the joy of JavaScript and TypeScript is the wide variety of the ecosystem and the freedom that this variety brings to software development.

Unfortunately, this variety also brings unexpected barriers which can be frustrating. Particularly when you're getting used to JavaScript/TypeScript, it can feel like you're fighting against your tools—particularly if your tech background includes technologies that Just Work™ more reliably than JS/TS.

As a result, we've put together this document with some extra tips to avoid some of that potential frustration as you get into the Mars Rover project!

2️⃣ The Environment

It is an important skill for JavaScript developers to be able to set up and configure their environments. This includes things like:

- Installing npm packages
- Updating dependencies and settings in `package.json`
- Getting the right settings in other configuration files, such as `tsconfig.json` (for TypeScript) or `.eslintrc` or many other examples
- Dealing with incompatible packages or settings
- Understanding how the `node_modules` folder is kept up to date with the dependencies specified in `package.json`
- Getting TypeScript to compile and run for **both** applications and testing

However! **We don't want you to fight with your environment setup during the Mars Rover.** Those skills can develop later—for now, it's more important to work on the actual code.

At this point, you ought to have seen an example repository which has everything "ready to go" for TypeScript—including 1️⃣ Jest for testing plus 2️⃣ compiling and running of the app via `npm start`.

👉 **Feel free to use that template repository as a template for your Mars Rover!**

👉 If you DON'T have access to such a repository, please ask us for one Slack!

> 💡 Take some time time to familiarise yourself with the `package.json` and `tsconfig.json` in the template, just so you have an idea of what a minimal working configuration looks like which includes TypeScript for both testing and application.

3️⃣ Project Structure

Depending on your coding background, it may not be obvious how to *actually* setup your Mars Rover. You may be tempted to create a file called `index.ts` so you can start coding and see what happens!

(Or, as this is TDD, start with `index.test.ts` at least 😅)

While that could work, it would be better to have more of a plan 🔬 so you don't end up with a big mess of code all in one file.

🔥 Where to put your files:

👉 Keep it simple! Put your code inside the `/src/` folder.

👉 Use subfolders if necessary. For example, you may end up with a handful of plateau-related files. If you want, you could group them in a folder: `/src/plateau`

🔥 What files to create:

👉 It's up to you, naturally! But part of the brief is to think of this as the beginnings of a potentially large application, so it makes sense to create some kind of initial structure.

👉 It's best to **group code by concern**. This means putting all related code together, which is helpful to navigate and understand large projects.

👉 As a result, it's likely you're going to need code for—at the very least—a Rover and a Plateau. So you can create `/src/rover.ts`, `/src/rover.test.ts`, `/src/plateau.ts` and `/src/plateau.test.ts` as a start!

👉 While you're coding, ask yourself if you're coding in the right file. For example, if you find yourself writing a whole bunch of code concerning compasses and directions - maybe that could be a separate module that your rover calls? It's up to you to make these decisions!

4 How to Begin

> *"Bad programmers worry about the code. Good programmers worry about data structures and their relationships.*
>
> *— Linus Torvalds"*

That quote is perhaps a little aggressive! But it became a popular quote because of the kernel of truth: **defining your data structures is incredibly important**.

To take an extreme example: Imagine I want to calculate something about a grid, but my grid object stores the grid data in a "*hexadecimal string that converts to the value of my X coordinate in Spanish*". It would be an absolute nightmare to make any calculations from this data model! My code will turn into a spaghetti-mess as I constantly convert back and forth from numbers to the weird data choice of my grid. 🍝

Instead, if I define my grid like this:

```
type Grid = { x: number, y: number };
```

Then calculations become easy, and straightforward. And recall that there are always alternative representations! For example, I could do this instead:

```
type Grid = [ number, number ];
```

💡 That second example uses a **Tuple** - which is like an array limited to a certain length. In this case, it's an array that is forced to have two elements which are both numbers. Learn more here:

https://www.w3schools.com/typescript/typescript_tuples.php

So, it's not the case of finding the *only* good data model for our problem. We just have to find *A* good data model for our problem.

👉 Begin by defining your types. **What data will a Rover need? What data will a Plateau need?**

👉 Aim to be specific. A Rover faces a particular direction, which could be a string… but `string` is very vague. "N" is a string, but so is "North" and so is "yoghurt". Restrict types wherever possible.

e.g.

```
function executeInstruction(instruction : string) {

}
```

could be:

```
function executeInstruction(instruction : 'L' | 'R' | 'M') {

}
```

```typescript
function executeInstruction(instruction : 'L' | 'R' |
'M') {

    // function body

}

const someInstruction = getInstruction(); // returns a
string

executeInstruction(someInstruction); // ⚠️ this will
error! 🥳
```

👉 We will look in detail at <u>type narrowing</u> later in the programme. For now, here's a handy snippet you could use to solve the error in the example above:

```typescript
function isInstruction(input: string)

    : input is 'L' | 'R' | 'M'

{

    return ['L', 'R', 'M'].includes(input);

}
```

💡 **Note the weird return type:** `instruction is 'L' | 'R' | 'M'`

A result from a function with the "is" keyword allows TypeScript to be sure that anything that comes through that function with a 'true' result fits that condition. We will look at this syntax in detail in future to understand it better!

Now we can fix the error in the example on the previous page:

```
const someInstruction = getInstruction(); // returns a string

if(isInstruction(someInstruction))

    executeInstruction(someInstruction);

    // 🥳 no error!
else

    // 😮 it's not an instruction! We should handle this case!
```

💡 **This** `isInstruction` **function is an example of a "type guard"**

🔥 Defining Types

👉 Note that if you're repeating yourself with a type you can define it once:

```
type RoverInstruction = "L" | "R" | "M";
```

All of the above examples could use this RoverInstruction type instead of the repetition of "L" | "R" | "M".

⚠️ A Warning for OOP Programmers

⚠️ If you're an OOP programmer—meaning your background is mostly in languages like C#, or Java, which are strongly *object-oriented*—then you may be tempted to immediately reach for a series of `class` keywords to solve all your problems. While it's okay to use classes in JS/TS, **it's more common to stick to basic JavaScript objects**.

👉 **For example:**

```
class Plateau {

    x: number = 0;
    y: number = 0;

    constructor(x: number, y : number){
        this.x = x; this.y = y;
    }
}
```

👉 **The above would be more common as a simple typed JavaScript object:**

```
type Plateau = { x: number, y: number };
```

💡 In general, prefer basic objects... BUT if you find yourself creating something that will have many instances, and it includes both state data and methods, then perhaps a `class` is appropriate! It's okay to mix-and-match as you see fit.

5 Other Considerations

🔥 Const arrays:

In JavaScript, declaring an array with const only means you can't reassign the variable. The contents of the array can still change.

If you want to ensure the contents can't change either, TypeScript lets you do this:

```
const SOME_FOODS = [ "Noodle", "Pasta", "Spaghetti"] as const;
```

You can then use this to derive a type:

```
type Food = typeof SOME_FOODS[number];
```

👉 This syntax is a bit weird-looking but we will go deeper into it later! But if you want to have a function that ONLY accepts foods defined in your array, you now can take a `Food` as a parameter and it will be either "Noodle", "Pasta" or "Spaghetti"

👉 You may have to combine this technique with the 'type guard' shown earlier. If you try to pass a variable with a `string` into a function that accepts only a `Food`, you have to prove to TypeScript—perhaps with a type guard—that that variable REALLY contains a `Food`.

⌨️ User Input

> ⚠️ **Be SURE to get all your basic logic working and well-tested with a full suite of automated unit tests BEFORE you think about adding UI!**

👉 Remember all your UI will come in as "string", whether you load it from a file or a console, so you'll need functions that check through any input and validates that it is one of your custom types, and responds appropriately if not.

👉 Make sure your logic doesn't depend on the UI! Your existing logic shouldn't need to change if you wrote it well in the first place.

👉 Keep the UI code physically separated in order to make it easier to prevent your logic from depending on your UI. Use this rule for yourself:

✅ Your UI files may `import` from your logic code

❌ Your logic files may NOT `import` from your UI code.

This Mars Rover brief was inspired by

[Mars Rover Kata](#).