simple-ts-sample

Release 0.1.0

Alessandra Bilardi

Mar 21, 2021

CONTENTS:

CHAPTER

ONE

TYPESCRIPT PROTOTYPE

This package contains a simple sample of a Typescript package prototype. It is part of the educational repositories to learn how to write stardard code and common uses of the TDD.

See the documentation and how to do it on readthedocs. And see the development of this code step by step

- with see-git-steps
- on readthedocs / step by step

1.1 Installation

The package is not self-consistent: it needs to install npm, the Node Package Manager.

And then you can download the package by github:

```
$ git clone https://github.com/bilardi/typescript-prototype
```

Or you can install by npm:

```
$ npm install simple-sample
```

1.2 Usage

Read the unit tests in tests/myClass.test.py file to use it. This is a best practice.

```
import { MyClass } from 'simple-sample';
console.log(new MyClass(true));
```

1.3 Development

It is common use to test the code step by step and unittest module is a good beginning for unit test and functional test.

Test with unittest module

```
$ cd typescript-prototype
$ npm run test # see package.json for other details
```

Test all the other steps

```
$ cd typescript-prototype
$ make ptest # see Makefile for other details
```

1.4 Change Log

See CHANGELOG.md for details.

1.5 License

This package is released under the MIT license. See LICENSE for details.

CHAPTER

TWO

HOW TO USE

In this section you can find some tips & tricks for learning to use any code.

Instead, if you need to learn typescript language, it is better to read something dedicated like a book or a rich documentation.

2.1 Unit tests

When you change something on your code, you can run one unit test about that class changed

```
$ cd typescript-prototype
$ npm test -- tests/myClass.test.ts
```

And when you are ready for the commit, you can use a command for running all unit tests

```
$ cd typescript-prototype
$ npm test
```

But for learning how to use an class in your code, you need to read its unit test file. You can find the import class, the initialization class, and the main public methods.

2.2 MyClass

The precedent approches are the best practice for learning something about a specific package.

Sometimes, the package is so complex, that it is also necessary a "Quick start" where a developer can learn the main classes or methods to start from

So it is a good practice to improve some lines: in this repo you can find a file in the folder example

```
import { MyClass } from 'simple-sample';
console.log(new MyClass(true));
```

For running the example, you can copy that file in **another folder** as described below

```
$ mkdir example
$ cp typescript-prototype/example/example.ts example/
$ cd example
$ npm install simple-sample
$ npx ts-node example.ts
```

CHAPTER

THREE

HOW TO MAKE

In this section you can find how to generate or publish that helps you with managing your code.

3.1 How we create it

These steps are only for education propose because you can clone this package for starting with your projects like the description in Getting started section.

The main commands that we used for create the package are

```
$ cd your-typescript-project
$ npm init -y
$ tsc --init
$ npm install --save-dev typescript # install TypeScript as a devDependency in␣
↪package.json
```

These commands create package.json and tsconfig.json files.

In this commit you can find a basic change: you can use these configuration files like your start.

The main configuration file is **package.json** that it could contains all you need:

- the main properties named **main**, **types** and scripts
- outline properties named **directories**, **engines**, **repositories** and **bugs**
- and then, some properties to improve your code: jest for your TDD and eslintConfig for linting

3.2 TDD

Before write code, it is important to verbalize the concepts by documentation and to create Test Driven Development (TDD) for your code. Then, it is important to use unit test for finding the issues and before to update change log file and package version.

See the development of this code step by step on readthedocs / step by step for learning how to make a unit test.

You can install your npm tests package using npm: below there is the installation command of the test framework used in this sample

```
$ cd your-typescript-project
$ npm install --save-dev jest @types/jest ts-jest jest-extended # install Jest as a␣
↪devDependency in package.json
```

3.3 Linting

When you write your code, you can use any editor: code editor or not, with autocompile or not, with plugin for improving your code or not.

The best practice is to use a static program analysis for identifing the main mistakes of form:

- if one variable is initialized with `let` but then it is not modified, you might consider using `const`
- if you have defined to use only doublequote, a static program analysis can help you for this

You can install your npm lint package using npm: below there is the installation command of the lint framework used in this sample

```
$ cd your-typescript-project
$ npm install --save-dev eslint @typescript-eslint/eslint-plugin@latest # install␣
↪ESlint as a devDependency in package.json
```

3.4 Scripts

In the file named **package.json** there is an important section: scripts.

In this section, you can define your specific commands that npm will run for you when you run

- a command with the format like `npm run <name-of-your-property>`
- a standard command for the operation on life cycle (see details in the scripts documentation page)

There are many methods to define what you want to do for each step of package life:

- you can use only the section **scripts**
- you can also use a **Makefile** so the section scripts will have kept slim
- another way it is to keep separated the section scripts and Makefile

It is a choice to force everybody uses your specific commands: in this repo, there are a Makefile uses the section scripts, so they are separated.

3.5 Packaging

The tutorial for packaging your projects is standard. And then your package is public on npm.

The main commands for testing your package,

```
$ cd typescript-prototype
$ npm link src
$ cd example
$ npm link simple-sample # locally simulates npm install
$ npx ts-node example.ts
$ cd -
```

It is a best practice to test everything before to publish, so this repo has a Makefile with some commands for testing all locally but not only:

```
$ make help # for printing the commands list
$ make itest # for testing all locally
```

If you have no problem, but this is a beta version, you can publish with

```
$ npm publish --beta
$ # or
$ make btest # run all tests and then publish a beta version of the package
```

If you have any issue, you can try to create the package locally to investigate the problem

```
$ run pack # locally generates a tarball of everything that will get sent to and␣
↪published on npm
$ # or
$ make ptest # run all tests and then generate the package
```

The main commands for publishing your package

```
$ npm publish
$ # or
$ make build # run all tests and then publish the package
```

CHAPTER

FOUR

STEP BY STEP

This page is for learning all steps that they are necessary for writing a simple package like simple-sample.

4.1 see-git-steps

The package see-git-steps has been written for reading piece by piece the commits of a git repository.

If you want to use it, you have to install it following its README.md.

4.2 Getting started

The goal of the package simple-sample is to create a TypeScript package prototype. So you can use this simple package for downloading a base for your package.

4.2.1 Step 1

The first step is to add all the outline files for your package

```
$ cd typescript-prototype
$ see-git-steps
179811d93b9fcc77c0fdb76ba53102cfdbe2e8d9 step 1 - add the outline files
.gitignore
.npmignore
CHANGELOG.md
LICENSE
MANIFEST.in
Makefile
README.md
README.rst
package-lock.json
package.json
tsconfig.json
```

They are important files and below you can find a link of a guide for each file

- .gitignore, to ignore specific files when you have to commit
- .npmignore, to ignore specific files when you have to publish the package
- CHANGELOG.md, the best practise for reading the minor o major change of your code
- LICENSE, the best practise for defining your policy for the public repository

- MANIFEST.in, documentation included into your package
- Makefile, it is not necessary but it is a comfortable way to remember procedures
- README.md, documentation visible on your repository homepage and package page
- README.rst, documentation visible on readthedocs
- package-lock.json, it describes the exact tree of the packages used
- package.json, it describes the requirements and behaviour of the package
- tsconfig.json, it describes the requirements to compile the package

The files that you have to customize are

- **.gitignore**, for example, if you use an ide with specific files extension
- **LICENSE**, with year and your name
- **Makefile**, with your PACKAGE_NAME and YOUR_USERNAME of npm
- **README.md / README.rst**, with your quick start documentation

For initializing a npm package,

```
$ cd typescript-prototype
$ npm init -y # create package.json
$ tsc --init # create tsconfig.json
$ npm install --save-dev typescript # install TypeScript as a devDependency in␣
↪package.json
```

When you have modified, you can commit your first change

```
$ cd typescript-prototype
$ git init # for initializing the repository
$ git add .gitignore CHANGELOG.md LICENSE MANIFEST.in Makefile README.md *json
$ git commit -m "step 1 - add the outline files"
```

4.2.2 Step 2

The second step is to custom the configuration files with what you need for creating your package

```
$ cd typescript-prototype
$ see-git-steps
f7c3e395376537c9abf1cd2cf778bce5d4de7854 step 2 - update configuration files
package-lock.json
package.json
tsconfig.json
```

For testing and linting your code, you have to install some packages

```
$ cd typescript-prototype
$ npm install --save-dev jest @types/jest ts-jest jest-extended # install Jest as a␣
↪devDependency in package.json
$ npm install --save-dev eslint @typescript-eslint/eslint-plugin@latest # install␣
↪ESlint as a devDependency in package.json
```

Those commands update the configuration files, and for packaging your code, you have to define some other variables

- **package.json**, where you find variables like name, description and scripts

- **tsconfig.json**, where you find variables like outDir, rootDir and exclude

See the changes these configuration files by GitHub or by command line with see-git-steps

```
$ cd python-prototype
$ see-git-steps -c f7c3e395376537c9abf1cd2cf778bce5d4de7854 -v
```

The file **package-lock.json** is automatically generated.

When you have modified the configuration files, you can commit your changes

```
$ cd typescript-prototype
$ git add *json
$ git commit -m "step 2 - update configuration files"
```

4.2.3 Step 3

Before write code, it is important to verbalize the concepts by documentation: so the documentation is important to learn a package as to plan how to write the code.

You can write your documentation as you want: you can create docs folder like in this package, by sphinx.

You can also write an example of code that it uses your future package, that you will use for testing each your new release.

When you have created your documentation, you can add the new folder and you can commit your changes

```
$ cd typescript-prototype
$ git add docs example
$ git commit -m "step 3 - add documentation by sphinx and example"
```

When a commit completes one feature or a set of fixies, you can tag that commit as a release. The standard behaviour is to add changes in a CHANGELOG file: see the changes of **CHANGELOG.md** by GitHub or by command line with see-git-steps

```
$ cd typescript-prototype
$ see-git-steps -c 0a6442f798934183b36167246eef5d103194b432 -v | head -n 34 | tail -n␣
↪17 # for CHANGELOG.md details
```

So you can add CHANGELOG.md on your last commit, or you can create one commit for changelog, and then you can add the tag.

```
$ cd typescript-prototype
$ git add CHANGELOG.md
$ git commit --amend # add file on your last commit
$ git tag v0.0.1 -m "Empty package and documentation by sphinx" # create a tag with␣
↪that version name
$ git tag -n # show the tag list with description
$ git push origin --tags # load the tag on repository
```

4.2.4 Step 4

Before write code, it is important to verbalize the methods by create Test Driven Development (TDD) for your code. Then, it is important to use unit test for finding the issues and before to update change log file and package version.

In TypeScript, there are many TDD frameworks: in this repo is used Jest.

Natively, Typescript does not allow to instantiate an abstract class or an interface.

So the tests are only on the final public functions: see the unit tests by GitHub or by command line with see-git-steps

```
$ cd typescript-prototype
$ see-git-steps -c 4bf512b2f38372cd2eb97b7006131dfb5dd62f98 -v
```

When you have created **tests/myClass.test.ts**, you can commit your changes

```
$ cd typescript-prototype
$ git add tests/myClass.test.ts
$ git commit -m "step 4 - add the unit test"
```

4.2.5 Step 5

Now you can write your first code: see myClassAbstract by GitHub or by command line with see-git-steps

```
$ cd typescript-prototype
$ see-git-steps -c 48a7b214cff13cd179dbdfa404895dc96521acfb -v
```

A class abstract cannot be instantiated, so you cannot test directly.

When you have created src/myClassAbstract.ts, you can commit your changes

```
$ cd typescript-prototype
$ git add src/myClassAbstract.ts
$ git commit -m "step 5 - add myClassAbstract"
```

4.2.6 Step 6

Now you can write your second code: see myInterface by GitHub or by command line with see-git-steps

```
$ cd typescript-prototype
$ see-git-steps -c f4c3acf94b79d4939436204cf30d5c116d98c553 -v
```

An interface can only be extended, so you cannot test directly.

When you have created src/myInterface.ts, you can commit your changes

```
$ cd typescript-prototype
$ git add src/myInterface.ts
$ git commit -m "step 6 - add myInterface"
```

4.2.7 Step 7

Now you can write your third code: see myType by GitHub or by command line with see-git-steps

```
$ cd typescript-prototype
$ see-git-steps -c ac1ae35f85bba8cab099dae166b8b4f772fa9a01 -v
```

A functional test for a type is to use it, so you can test directly on myClass.

When you have created src/myType.ts, you can commit your changes

```
$ cd typescript-prototype
$ git add src/myType.ts
$ git commit -m "step 7 - add myType"
```

4.2.8 Step 8

After verbalizing all functions by the unit tests, after to write myClassAbstract, myInterface and myType, you are ready to write myClass and the index: see new files by GitHub or by command line with see-git-steps

```
$ cd typescript-prototype
$ see-git-steps -c fce06f494a319e2a282d282e1c5905941db61505 -v
```

When you have created new files, you can run all unit tests

```
$ cd typescript-prototype
$ npm run test

> simple-sample@0.0.1 test
> jest

PASS  tests/myClass.test.ts
Simple tests
    ✓ Passes when MyClass can be instantiated (3 ms)
Functional tests
    ✓ Passes when baz returns a boolean
    ✓ Passes when foo returns the reverse value of its input (1 ms)
    ✓ Passes when getBar returns the value of bar
    ✓ Passes when fooBar returns the reverse value of bar (1 ms)
    ✓ Passes when fooQuux returns the reverse value of qux
    ✓ Passes when getGrault returns the value of corge (2 ms)
    ✓ Passes when fooGrault returns the reverse value of corge (1 ms)

Test Suites: 1 passed, 1 total
Tests:       8 passed, 8 total
Snapshots:   0 total
Time:        1.708 s
Ran all test suites.
```

If all test is OK, you can add the new file and you can commit your changes

```
$ cd typescript-prototype
$ git add src/myType.ts
$ git commit -m "step 8 - add index, myClass and unit tests works properly"
```

to be continued ..

CHAPTER

FIVE

INDICES AND TABLES

- genindex
- modindex
- search