Starting a Typescript CLI Project from Scratch

Adrian Kosmaczewski

2020-12-11

The JavaScript ecosystem has grown dramatically in the past decade. It has become so complex, that I've seen many new developers interested in the subject struggle to find out where to start.

I decided to publish a blog post with the required steps to specifically bootstrap a command-line project with TypeScript. This provides a simple, straightforward, very opinionated, tried and tested approach to create a new project with many features required by rational software development processes.

Opinion

I think the tooling around JavaScript has improved a lot lately; it has become a serious contender for building all kinds of applications, and people are using it pretty much for anything and everything these days.

But it remains a complex beast to tame, and in many cases I can hardly recommend all of the frameworks available. Falling in this category I can mention the Ionic Framework, for example. Stay away from it if you can. I prefer to focus

this blog post into the things that I found actually bring value to projects. These are more related to good practices, like static analysis, testing, documentation, etc.

Having said that, let us state clearly the first factor of success for any JavaScript project: to *not* use JavaScript at all, but rather TypeScript instead.

TL;DR: Cookiecutter

Because life is too short to read whole blog posts, I've created a Cookiecutter with the final Node.js project; feel free to create your new TypeScript CLI projects from scratch using the following command:

```
$ cookiecutter https://gitlab.com/akosma/typescript-cli-cookiecutter
```

If you want to learn how that Cookiecutter was built, keep reading.

Requirements

Make sure you have the following software available before starting:

1. Terminal
2. Visual Studio Code
3. Usual command line tools: `git`, `curl`, etc.

Install Node.js

I never use the standard version of Node.js bundled with my system, if any, nor I install it following the usual advice you find online. Instead I always use nvm to manage my Node.js installations. This way I can have several Node.js versions installed, and I can switch from one to the other with one command. This is particularly useful if you work as a consultant, and you need different versions for different customers.

As an analogy, `nvm` is the equivalent in Node.js to pyenv for Python and rbenv for Ruby.

To install `nvm`, just run the following command:

```
$ curl -o- https://raw.githubusercontent.com/nvm-sh/nvm/v0.37.2/install.sh | bash
```

Then setup your environment in the terminal, adding these lines to your `~/.profile` or `~/.zshrc` file:

```
export NVM_DIR="$([ -z "${XDG_CONFIG_HOME-}" ] && printf %s "${HOME}/.nvm" || printf %s "${X
[ -s "$NVM_DIR/nvm.sh" ] && \. "$NVM_DIR/nvm.sh" # This loads nvm
```

Find the latest versions of Node.js, and install one of them:

```
$ nvm ls-remote | grep "Latest LTS"
$ nvm install 14.15.1
```

I like to install the latest LTS release; please bear in mind that the version numbers will have most probably changed by the time you read this.

Test your current installation with these commands; each output should point to an executable in `~/.nvm/versions/node/v14.15.1/bin`

```
$ which node
$ which npm
$ which npx
```

If you're using Windows, instead of `which` use the `where` command.

We're ready to go now.

New TypeScript Project

Now that we have Node.js installed, we can create a new project:

```
$ mkdir -p project/src
$ cd project
$ npm init
```

At this point you can just accept all defaults. You will find a new file called `package.json` in your project, used by Node.js and `npm` to store information about your project, including dependencies and all metadata.

Let's install TypeScript now:

```
$ npm install typescript ts-node --save-dev
```

This command will create a new folder in your project, the dreaded `node_modules` behemoth, which includes all the dependencies of your project. The `npm install` command has been the source of countless memes in the community.

The `ts-node` project allows you to run TypeScript code on the terminal, compiling it to JavaScript on the fly. This is super useful for CLI apps like the one we're building here.

Now we need to initialize our TypeScript installation with a `tsconfig.json` file:

```
$ npx tsc --init
message TS6071: Successfully created a tsconfig.json file.
```

The `npx` command executes binaries installed in your `./node_modules/.bin` folder. This helps you to avoid having to install commands globally, and will also help later on, in particular for building the application in different environments, such as during the creation of Docker containers.

I recommend to always install all of the tools required for a project locally, and never globally.

You can customize your `tsconfig.json` file, removing all comments and uncommenting the two lines shown below, so that it looks like this:

```
{
  "compilerOptions": {
    "target": "es5",
    "module": "commonjs",
    "sourceMap": true,  // Uncomment this
    "outDir": "out",    // and uncomment and set this
    "strict": true,
    "esModuleInterop": true,
    "skipLibCheck": true,
    "forceConsistentCasingInFileNames": true
  }
}
```

Write Code

Finally! The interesting part of our application. We are going to use Visual Studio Code to create a class and a program consuming it. I chose Visual Studio Code because it comes from the same people who created TypeScript, and it has stellar support for it.

```
$ code .
```

Create two TypeScript files in the `src` folder; first, `src/customer.ts`:

```
export class Customer {
  readonly name : string

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

  greet(): string {
      return `Hello ${this.name}`
  }
}
```

This is a very simple TypeScript class, with a constructor and a method called `greet()`. Nothing to phone home about. You will add in this folder your own logic, as required by your own needs.

Then we will create the entry point of our application, `src/index.ts`:

```
import { Customer } from './customer'

const cust = new Customer('Toto')
console.log(cust.greet())
```

You can now test the application by running this command:

```
$ npx ts-node src/index.ts
Hello Toto
```

Tada! Your application is running, greeting through an instance of the `Customer` class.

We can update the `package.json` file now, including some tasks:

```
{
  "name": "project",
  "version": "1.0.0",
  "description": "",
  "main": "src/index.ts",       // Change this if needed
  "scripts": {
    "build": "npx tsc",         // Add this line
    "test": "echo \"Error: no test specified\" && exit 1"
  },
  "author": "",
  "license": "ISC",
  "devDependencies": {
    "ts-node": "^9.0.0",
    "typescript": "^4.0.3"
  }
}
```

The changes above allow us to type the following commands:

```
$ npm run build
$ node out/src/index.js
Hello Toto
```

The entries in the `scripts` section of the `package.json` file allow us to use `npm` as if it were a `make` command.

Clean Task

It is a good practice to have a `clean` task to quickly remove all build products. We could add a line like this to our `package.json` to do that:

```
  "scripts": {
    "build": "npx tsc",
    "release": "npx gulp",
    "clean": "rm -rf out", // this line
    "test": "npx ts-node node_modules/.bin/jasmine spec/*"
  },
```

But the problem is that the `rm -rf` command does not work in Windows (and yes, I care about our poor Windows users). Thankfully there's a simple

workaround to this issue:

```
$ npm install rimraf --save-dev
```

Now we can use the `rimraf` command, which works everywhere:

```
  "scripts": {
    "build": "npx tsc",
    "release": "npx gulp",
    "clean": "npx rimraf out", // better!
    "test": "npx ts-node node_modules/.bin/jasmine spec/*"
  },
```

And now we can clean our project simply by doing

```
$ npm run clean
```

Writing TypeScript

I will not explain *all* of TypeScript in this tutorial, but just provide a cheatsheet of the most important things you need to know about it:

- All JavaScript is valid TypeScript. Not all valid TypeScript is valid JavaScript (yet).
- All JavaScript numbers are floats. There are no integers. Pay attention to rounding errors.
- Use `const` and `let` instead of `var`.
- Use `for of` instead of `for in`.
- Functions are first-class objects.
- String interpolation *and* multi-line strings use backticks: `Some text and a ${variable}`.
- Forget about semicolons; the compiler will add them.
- Add comments, lots of comments; the compiler will strip them.
- Surround strings with 'single quotes'. This is particularly useful to output strings that contain double quotes, like in HTML.
- Variables and classes in TypeScript "work as expected", unlike regular JavaScript.
- There is a module system similar to that of Python.
- The TypeScript type system is outstanding: type inference, interfaces, classes, enums, generics with constraints, literal types, union and intersection types, nullable types, type aliases, structural types, function decorators…
- There are `@types/xxxxx` packages available through `npm`, which provide type information for many popular JavaScript packages, so that you get a better developer experience. Not all packages have one, but most do.
- Use type `any` *only* to interact with existing JavaScript; always type variables and return values explicitly in your own code.
- Prefer inline functions `() => {}` to classic `function` keyword.
- Make numbers more readable with the thousands separator: `1_000_000`.

- All JavaScript runs interpreted and single-threaded, in a single event loop. Multiprocessing is achieved through callbacks.
- Use `await` / `async` and `Promises` instead of "callback hell". This advice also includes using promises in the `fs` package.

Add Unit Tests

Mocha is a popular JavaScript testing library. Chai is an assertion library with various styles (should, assert, expect, etc). The `ts-mocha` project provides a way to directly execute Mocha tests written in TypeScript.

Install Mocha and Chai in your project, including the TypeScript type information for each:

```
$ npm install mocha @types/mocha ts-mocha chai @types/chai --save-dev
```

Create a file called `spec/customer.ts` to hold the tests:

```
import { expect } from 'chai'
import { Customer } from '../src/customer'

describe('A Customer', () => {
  it('greets', () => {
    const cust = new Customer('Toto')
    expect(cust.greet()).to.equal('Hello Toto')
  })
})
```

Modify `package.json` to add a test command

```
  "scripts": {
    "build": "npx tsc",
    "clean": "rimraf out",
    "test": "npx ts-mocha spec/*" // Add this line
  },
```

Run `npm test` or `npm run test`:

```
> project@1.0.0 test /home/akosma/Dropbox/Current/typescript-project
> npx ts-mocha spec/*


  A Customer
     greets


  1 passing (3ms)
```

For convenience, I suggest adding the Mocha Test Explorer in Visual Studio Code to have integrated testing support. This extensions uses another one from the same author, which works with many other programming languages.

By the way, if you are a fan of JetBrains IDEs, you should know that Mocha is also natively supported in WebStorm.

Git

At this point we should add this project to source control. Create a simple `.gitignore` file with the following contents

```
node_modules
out
dist
```

Run the usual commands to store everything properly:

```
$ git init
$ git add .
$ git commit -m "First commit"
```

Much better.

Gulp

Gulp is a very common JavaScript build tool. It reads tasks from a `gulpfile.js`, just like `make` would use a `Makefile`, or `ant` would use its `build.xml`.

This `gulpfile.js` is intended for command-line apps; for client web apps, one must use other libraries, such as `browserify` and `tsify`, which require a slightly different setup.

Bear in mind that this `gulpfile.js` provides uglification. This not only provides some obfuscation of your code, it actually helps bootstrapping the code faster in the runtime.

Install the required dependencies:

```
$ npm install gulp gulp-typescript gulp-uglify gulp-chmod gulp-insert @types/node --save-dev
```

Create `gulpfile.js` at the root of the project:

```
var gulp = require('gulp')
var ts = require('gulp-typescript')
var uglify = require('gulp-uglify')
var chmod = require('gulp-chmod')
var insert = require('gulp-insert')
var tsProject = ts.createProject('tsconfig.json')

function build() {
  return gulp.src('src/*.ts')
    .pipe(tsProject())
    .pipe(uglify())
    .pipe(gulp.dest('dist'))
```

```
}

function release() {
  return gulp.src('dist/index.js')
    .pipe(insert.prepend(`#!/usr/bin/env node
`))
    .pipe(chmod(0o755))
    .pipe(gulp.dest('dist'))
}

exports.default = gulp.series(build, release)
```

Pay attention to the line in the `release()` task, with the new line character at the end! That string is surrounded by backticks (“‘”) so that we can use a multiline string.

Modify `package.json` to add a new task, and a new folder to clean at the end:

```
  "scripts": {
    "build": "npx tsc",
    "release": "npx gulp",      // Add this line
    "clean": "rimraf out dist", // Add another folder to clean
    "test": "npx ts-mocha spec/*"
  },
```

Run the new command to drive `gulp` and build the application:

```
$ npm run release
$ dist/index.js
Hello Toto
```

Debug in Visual Studio Code

Instead of adding `console.log()` statements all over the place, we’d better use a debugger. We are going to configure Visual Studio Code for that.

In Code, use the `SHIFT + CTRL + D` shortcut (or its equivalent in the Mac) to open the Debug explorer. Click on the “create a launch.json file” link below the blue “Run and Debug” button of the debug explorer.

Select Node.js in the menu that appears.

Add the `preLaunchTask` element below in the `launch.json` file, and modify the entry point in the `program` element:

```
{
    "version": "0.2.0",
    "configurations": [
        {
            "type": "node",
```

```
            "request": "launch",
            "name": "Launch Program",
            "skipFiles": [
                "<node_internals>/**"
            ],
            "program": "${workspaceFolder}/src/index.ts",
            "preLaunchTask": "tsc: build - tsconfig.json",   // Add this line
            "outFiles": [
                "${workspaceFolder}/**/*.js"
            ]
        }
    ]
}
```

Open `src/index.ts` and place a breakpoint (or use the `debugger` keyword). Hit the `F5` key, select "Node.js" in the menu, and watch the breakpoint hit.

If it didn't open automatically, open the View menu, select "Debug Console" (`SHIFT + CTRL + Y`) and see the execution of the code.

Add External Libraries

The JavaScript ecosystem is flooded with libraries, of various degrees of quality. In this section we're just going to add one, and use it in our code.

Install the library locally, this time without the `--save-dev` parameter, as we want this library to be available during runtime, and not just at build time:

```
$ npm install cowsay
```

Execute `npm audit fix` if needed.

Modify `src/customer.ts` to use the library:

```
var cowsay = require('cowsay')

export class Customer {
  readonly name: string

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

  greet(): string {
    return cowsay.say({ text: `Hello ${this.name}` })
  }
}
```

Test run the code:

```
$ npx ts-node src/index.ts
 ------------
< Hello Toto >
 ------------
        \   ^__^
         \  (oo)\_______
            (__)\       )\/\
                ||----w |
                ||     ||
```

Fix the unit test (remember to use double backslash `\\` to escape the single backslash, and make the test pass):

```
import { expect } from 'chai'
import { Customer } from '../src/customer'

describe('A Customer', () => {
  it('greets', () => {
    const cust = new Customer('Toto')
    expect(cust.greet()).to.equal(` ------------
< Hello Toto >
 ------------
        \\   ^__^
         \\  (oo)\\_______
            (__)\\       )\\/\\
                ||----w |
                ||     ||`)
  })
})
```

Run the tests now, they should pass:

```
$ npm test
```

API Documentation

Now that we have a nice application, we need to document its APIs so that other developers can use our code.

Open the `src/customer.ts` file and type `/**` at the beginning of the `greet()` method. Hit `ENTER` and start writing the API documentation; it does not matter, just write something.

Open `src/index.ts`, and hover the mouse on top of the call to the `greet()` method to see the API docs in use.

You can export the API documentation into a nice HTML website using TypeDoc, which is the standard API documentation tool for TypeScript.

```
$ npm install typedoc --save-dev
$ npx typedoc --out docs -theme minimal src
```

Open `docs/index.html` to see the generated documentation.

Add `docs` to `.gitignore`. We don't want to store the generated documentation folder in Git.

```
node_modules
out
dist
docs
```

Edit `package.json` to add a `scripts` entry:

```
  "scripts": {
    "build": "npx tsc",
    "release": "npx gulp",
    "clean": "rimraf out dist docs", // Add a new folder to clean
    "docs": "npx typedoc --out docs -theme minimal src", // Add this line
    "test": "npx ts-mocha spec/*"
  },
```

ESLint

TypeScript ESLint is a project that provides a linter for TypeScript code. Linters perform static analysis of your code, showing you areas of improvement, and even highlighting potential security flaws. I strongly recommend to use a linter for your project, whatever programming language you use.

Install TypeScript ESLint:

```
$ npm install eslint typescript @typescript-eslint/parser @typescript-eslint/eslint-plugin -
```

Create an `.eslintrc.js` file in the root of the project with these contents:

```
module.exports = {
  root: true,
  parser: '@typescript-eslint/parser',
  plugins: [
    '@typescript-eslint',
  ],
  extends: [
    'eslint:recommended',
    'plugin:@typescript-eslint/recommended',
  ],
};
```

Create an `.eslintignore` file to avoid linting many folders in the project:

```
node_modules
out
dist
docs
```

Run the linter:

```
$ npx eslint src

~/project/src/customer.ts
  1:1   error  Unexpected var, use let or const instead        no-var
  1:14  error  Require statement not part of import statement  @typescript-eslint/no-var-req

 2 problems (2 errors, 0 warnings)
  1 error and 0 warnings potentially fixable with the `--fix` option.
```

Another benefit is that Visual Studio Code automatically detects the presence of ESLint and highlights the issues in our code.

We can run the command with the `--fix` option, which will remove one of the errors:

```
npx eslint src --fix

~/project/src/customer.ts
  1:16  error  Require statement not part of import statement  @typescript-eslint/no-var-req

 1 problem (1 error, 0 warnings)
```

The remaining problem has to do the `require` statement; in TypeScript it is preferred to use the `import` statement instead, but the `Cowsay` library unfortunately does not include a `@types/cowsay` library with the type information. At the time of this writing there's an open pull request waiting to be merged that precisely solves this issue.

In this case we will add a statement in our code to prevent ESLint from complaining about this fact:

```
/* eslint-disable @typescript-eslint/no-var-requires */
const cowsay = require('cowsay')

export class Customer {
  readonly name: string

  /**
   * This is a Customer, the most important thing, ever.
   */
  constructor(name: string) {
    this.name = name
  }
```

```
  /**
   * This method makes the customer greet.
   */
  greet(): string {
    return cowsay.say({ text: `Hello ${this.name}` })
  }
}
```

This will silence the linter until you find a better solution, which in this case would involve writing a types definition file for `cowsay` or waiting for the pull request to be merged.

Finally, as usual, add a command to your `scripts` section of the `package.json` file:

```
  "scripts": {
    "build": "npx tsc",
    "release": "npx gulp",
    "clean": "rimraf out dist docs",
    "docs": "npx typedoc --out docs -theme minimal src",
    "lint": "npx eslint src", // Add this line
    "test": "npx ts-mocha spec/*"
  },
```

Docker Container

Docker containers are very useful ways to encapsulate command-line applications, making it much simpler to integrate in CI/CD build systems, or to distribute to colleagues.

Create a `Dockerfile` for your application

```
# Step 1: Builder image
FROM node:14.15.1-alpine3.12 AS builder
COPY [".eslintrc.js", ".eslintignore", "tsconfig.json", "gulpfile.js", "package.json", "pack
COPY src /command/src
COPY spec /command/spec
WORKDIR /command
RUN npm install
RUN npm audit fix
RUN npm test
RUN npm run lint
RUN npm run release

# After the build, this only installs the libraries used in production,
# not the ones installed with the `--save-dev` parameter
RUN rm -rf node_modules
```

```
RUN npm install --production

# Step 2: Runtime image
FROM astefanutti/scratch-node:14.14.0
COPY --from=builder /command/node_modules /app/node_modules
COPY --from=builder /command/dist/*.js /app/
ENTRYPOINT ["node", "/app/index.js"]
```

In step 1 we execute the tests and linter; in case of an error in either case, the build process will stop automatically.

The runtime image used in step 2 is provided by Antonin Stefanutti from Red Hat, and provides a lightweight base image for Node.js applications. This way our final image is 41 MB, instead of the… 960 MB of the standard Node.js image, or the 122 MB of the `node:alpine` image used for the build process in step 1. Quite a difference.

Run the following commands using either Docker or Podman:

```
$ podman build -t typescript-tutorial .
```

Once the image is built, you can run it directly:

```
$ podman run --rm typescript-tutorial
```

And you can see it in your local registry, and remove it if needed:

```
$ podman images | grep typescript-tutorial
$ podman rmi typescript-tutorial
```

From this point on, you can share this image with others through a registry, like Docker Hub, Quay, or private registries such as Harbor, GitLab, or OpenShift.

What Next?

Here we are. We have a fully functioning TypeScript CLI project, ready to receive your own code and tests. It can be statically analyzed, built as an OCI container image, and a size-optimized one for that matter.

What else can you do now with your project? Although I've kept the examples and workflows as simple as possible in this text, there's a lot of extension points for your application:

1. Use the pkg module to encapsulate code in a portable executable, ready to be distributed without fear of dependencies.
2. Expand `.gitignore` with https://gitignore.io/api/node
3. Configure WebStorm to work with this project (and adapt `.gitignore` accordingly).
4. Add logging with winston.
5. Add command-line argument parsing with commander or with oclif.
6. Use `Promise` and `await` / `async` for asynchronous code.

7. Strip `alert()` and `console.log()` calls at build time with gulp-strip-debug.
8. Create a desktop application and distribute it with Electron.
9. Manipulate date and time information with Moment.js
10. Run shell commands from your own code using shelljs
11. Create PDFs with PDF-lib
12. Style `console.log()` to your liking