Making PT accessible

Implementing PT in TypeScript

Petter Sæther Moen

Thesis submitted for the degree of
Master in Informatics: Programming and System Architecture
60 credits

Department of Informatics
Faculty of mathematics and natural sciences

UNIVERSITY OF OSLO

Autumn 2020

Making PT accessible

Implementing PT in TypeScript

Petter Sæther Moen

Abstract

Acknowledgements

Contents

List of Figures

List of Tables

Preface

Part I

Introduction

Chapter 1

Introduction

1.1 What is PT?

1.2 Purpose of implementing PT in TS

Chapter 2

Background

2.1 Package Templates

2.2 TypeScript

Part II

The project

Chapter 3

Planning the project

3.1 What Do We Need?

- The ability to add custom syntax (access to the tokenizer / parser)
- Some semantic analysis.

3.2 Syntax

For the implementation of PT we need some syntax for the following:

- Defining packages (`package` in PTj)
- Defining templates (`template` in PTj)
- Instantiating templates (`inst` in PTj)
- Renaming classes (=> in PTj)
- Renaming methods (-> in PTj)

`template` and `inst` are both not used or reserved in the ECMAScript standard or in TypeScript, and can therefore be used in UKenglishwithout any issues.

The keyword `package` in TS / ES is as of yet not in use, however the ECMAScript standard has reserved it for future use. In order to "future proof" our implementation we should avoid using this reserved keyword, as it could have some conflicts with a potential future implementation of packages in ECMAScript. It could also be beneficial to not share the keyword in order to not create confusion between ES packages and PT Packages. `module` is also a keyword that could be used to describe a PT package, however this is also reserved in the ES standard, and should therefore also be avoided for similar reasons to `package`, to avoid confusion. We will therefore use (`pack` or `bundle`?) instead.

For renaming classes PTj uses =>, however in ES this is used in arrow-functions[**arrowfunction**]. To avoid confusion and a potentially ambiguous grammar we will have to choose a different syntax for renaming classes. PTj, for historical purposes, uses a different operator (`->`) for renaming class methods, however for keeping UKenglishsimple we will stick to only having one common operator for renaming.

ECMAScript currently supports renaming of destructured fields using the `:`(colon) operator and aliasing imports using the keyword `as`. Even though we opted to choose a different keyword for packages, we will here re-use the already existing `as` keyword for renaming

3.3 Why TypeScript

3.4 Choosing the right approach

Before jumping into a project of this magnitude it is important to find out what approach to use. The end goal of this project is to extend TypeScript with the Package Templates language mechanism, this can be achieved as following:

- Making a fork of the TypeScript compiler
- Making a preprocessor for the TypeScript compiler
- Making a compiler plugin / transform
- Making a custom compiler from scratch

3.4.1 TypeScript Compiler Fork

Possible, however not as accessible as other alternatives and will make upkeep expensive.

3.4.2 Preprocessor for the TypeScript Compiler

A lot more work than ex plugin / transformer.

3.4.3 TypeScript Compiler Plugin / Transform

As of the time of writing this the official TypeScript compiler does not support compile time plugins. The plugins for the TypeScript compiler is, as the TypeScript compiler wiki specifies, "for changing the editing experience only"[1]. However there are alternatives that do enable compile time plugins / transformers;

- ts-loader[**tsloadergithub**], for the webpack ecosystem
- Awesome Typescript Loader[**awesometypescriptloadergithub**], for the webpack ecosystem. Deprecated
- ts-node[**tsnodegithub**], REPL / runtime

- ttypescript[**ttypescriptgithub**], TypeScript tool TODO: Les mer på dette
- A compiler wrapper

Unfortunately most of these don't support custom syntax, which is one of the requirements we have in order to properly implement PT.

3.4.4 Babel plugin

Babel isn't strictly for TypeScript, but for JavaScript as a whole. Will however make it very accessible as most web-projects use Babel, and the upkeep is cheap, as plugins are pretty independent from the core.

Saying that this is strictly a Babel plugin wouldn't be entirely true, as we would have to fork the Babel parser in order to include our custom syntax[**babelparserdocs**]. However this is all hidden away from the user, as this custom parser is a dependency of our Babel plugin.

Babel Plugin is not as nice as I first thought, it seems to be pretty hard to write third-party plugins as you have to make a parser fork for custom syntax, and there is a severe lack of documentation. Most examples of Babel plugins mostly use internal helpers and utils, which are hard to use for third-party plugins.

TODO: Does it support having multiple custom parser? E.g. babel-plugin-typescript + our custom babel plugin?

Part III

Conclusion

15

Chapter 4

Results

Bibliography

[1] Microsoft. *microsoft/TypeScript*. URL: https://github.com/microsoft/TypeScript/wiki/Writing-a-Language-Service-Plugin.