Vincent Bailly

Follow 17 Followers About

Evolving a large TypeScript repository

Vincent Bailly Jul 9, 2019 · 9 min read

Photo by niko photos on Unsplash

Midgard is a Microsoft internal git repository hosting code for high-value front-end components consumed throughout Microsoft 365. It is a mono-repo in the sense that the code is divided into multiple packages. It currently has around 1M lines of code, most of which is TypeScript (TS). This code is divided into around 100 NPM packages. In this article, we will see how a small change in the tooling may greatly enhance the developer experience.

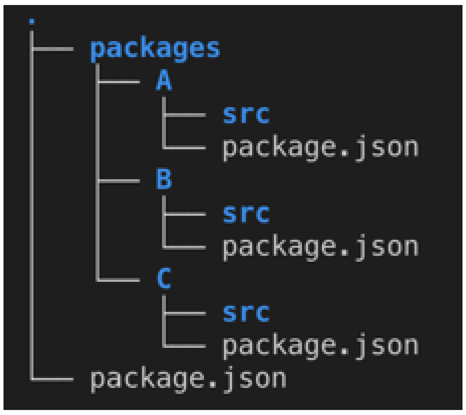
We chose to co-locate various components in one repository to foster collaboration, encourage code sharing and make cross-package development agile. To achieve these goals, it is important that the incentives for coding in Midgard vastly outweigh the incentives for creating a new repository. As guiding principle, we want the developer experience to be reliable, fast and understandable.

In this article, we will mention many times VSCode IntelliSense. This is a tool we use a lot in Microsoft to improve developer experience. Everything said about it can be applied to other text editors supporting TypeScript because they all use the same language server under the hood.

. . .

The current layout of Midgard is demonstrated in this sample repository.

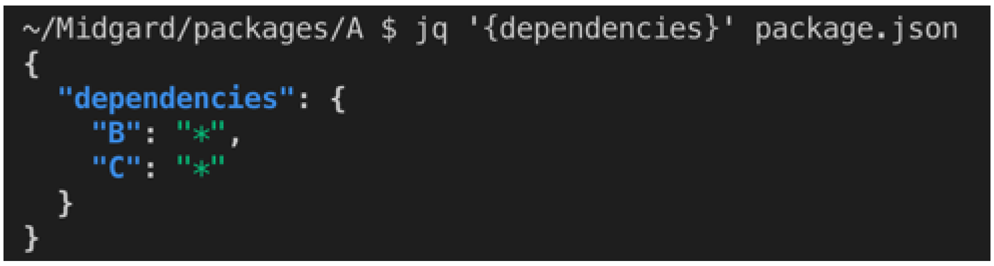
It is a flat list of packages in the same directory. Each package has a package.json file as you would expect from an NPM package.



```
.
├── packages
│   ├── A
│   │   ├── src
│   │   └── package.json
│   ├── B
│   │   ├── src
│   │   └── package.json
│   └── C
│       ├── src
│       └── package.json
└── package.json
```

High level layout of Midgard

The dependencies between packages are expressed in the package.json files.



```
~/Midgard/packages/A $ jq '{dependencies}' package.json
{
  "dependencies": {
    "B": "*",
    "C": "*"
  }
}
```

Midgard dependencies expressed in the package.json file

We use yarn workspaces to install our external dependencies and link the Midgard-hosted packages together based on the dependencies expressed in the package.json files.

Midgard-hosted packages are consumed both by other Midgard-hosted packages and by other Microsoft repositories. Even though our packages contain TS code, their dependents don't consume TS code but JavaScript (JS) code. The JS files exposed by a package to it dependents are reflecting the file and folder structure of its TS files. In other words, one TS file produces one JS file. The conversion from TS to JS is done by the TypeScript compiler (TSC).

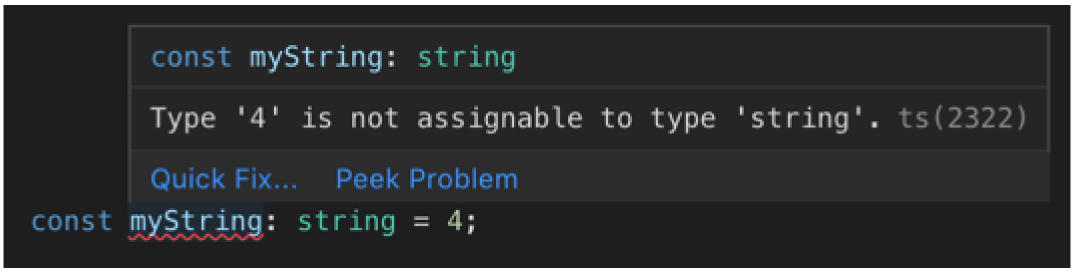
Ultimately, our code is shipped to our users in the form of bundles. A bundle is a standalone JS file that contains all the code needed to render a given component, including all its dependencies. Some bundles are created by other repositories depending on Midgard while other bundles are directly created in Midgard. The bundles created in Midgard are built using Webpack. The input for the bundling process is the JS files produced by the TSC process.

. . .

Since this article is mainly about developer experience, let's have a look at the main Midgard workflows.

When we fix a bug or create a new feature, we do the code change and then validate that it is correct. We can validate our code by deploying it to a production-like environment, but because this process takes around 45 minutes, we have in addition shorter workflows designed to help us validate our code faster.

The fastest validation is type-checking; when we produce invalid TypeScript code, IntelliSense gives us instant visual feedback within the editor. This is probably the most useful and loved validation step. So we want to make sure it works and it works fast.



```
const myString: string
Type '4' is not assignable to type 'string'. ts(2322)
```

```
const myString: string = 4;
```

IntelliSense informing about a type mistake.

A slightly slower validation workflow is unit testing. We won't talk about it more in this article.

To validate more complex component behaviors, like user interactions, we use test-apps. Test apps are either webpages or native applications which load our bundles to render our components for manual testing. This is probably the second most popular validation workflow, therefore we care a lot about it.

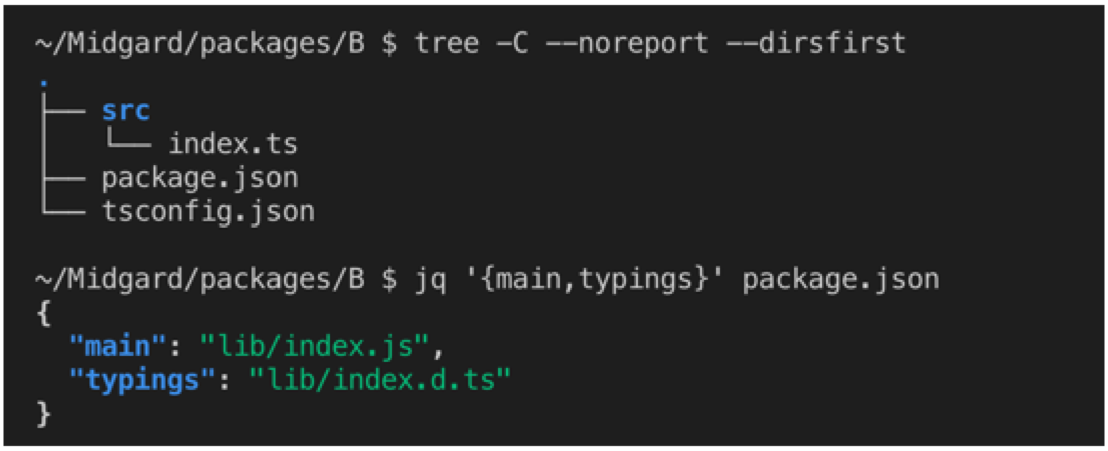
To validate the integration within the various applications, we publish the code to a production-like environment in which we can manually run some tests.

And finally, we can validate risky changes by running users workflows in a production-like environment in an automated fashion. This validation strategy uses screenshot diffing. We won't talk about it more in this article.

. . .

While there is a lot that can be done to improve the developer experience, this article focuses on what we think is the bottleneck for developer productivity. To understand this bottleneck, we need to explore the inner workings of a Midgard-hosted package.

On a clean repository, a typical package has its TS files in an "src" folder.

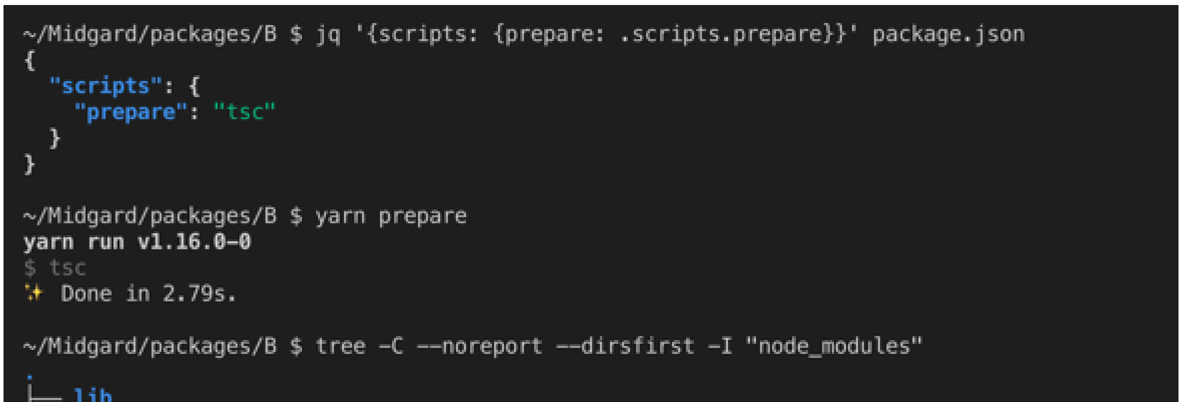


```
~/Midgard/packages/B $ tree -C --noreport --dirsfirst
.
├── src
│   └── index.ts
├── package.json
└── tsconfig.json

~/Midgard/packages/B $ jq '{main,typings}' package.json
{
  "main": "lib/index.js",
  "typings": "lib/index.d.ts"
}
```

Typical Midgard-hosted package layout on a clean repository.

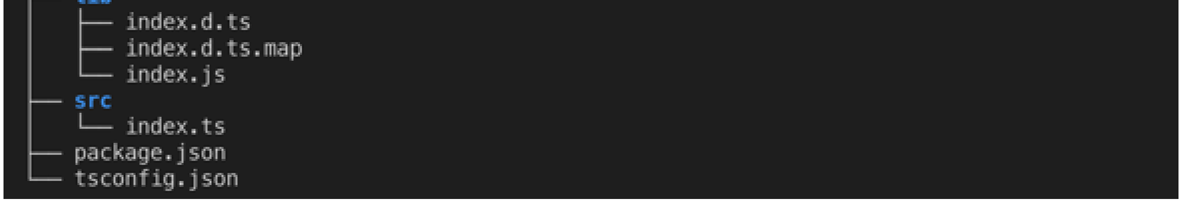
In the package.json file, the fields "main" and "typings" describe the entry points of the package public interface. We can see in the screenshot that these files don't exist on a clean repository; they need to be generated before this package could be consumed. This generation is done by invoking the TypeScript compiler.



```
~/Midgard/packages/B $ jq '{scripts: {prepare: .scripts.prepare}}' package.json
{
  "scripts": {
    "prepare": "tsc"
  }
}

~/Midgard/packages/B $ yarn prepare
yarn run v1.16.0-0
$ tsc
✨ Done in 2.79s.

~/Midgard/packages/B $ tree -C --noreport --dirsfirst -I "node_modules"
.
└── lib
```



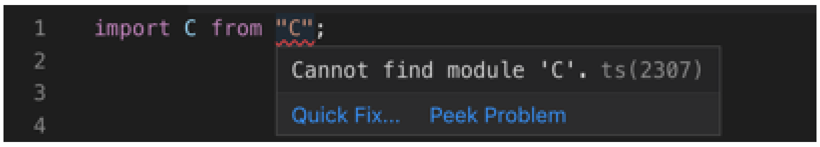
```
│   ├── index.d.ts
│   ├── index.d.ts.map
│   └── index.js
├── src
│   └── index.ts
├── package.json
└── tsconfig.json
```

Preparing a package for consumption with TSC

TSC will produce the files described in the package.json file, so now the package is ready to be consumed. The *.js files are the JS files containing the code ready to be consumed. The *.d.ts files (DTS) are the files containing type information, they are used to type-check the code consuming this package. The *.d.ts.map are used by IntelliSense to enable cross-package code navigation. Because producing these files is necessary before a package can be consumed, we call this step “prepare”.

To make it clear why we need the “prepare” step before edition the code, let’s look a the IntelliSense experience in a package importing package C in two cases: when C is not prepared and when C is prepared.

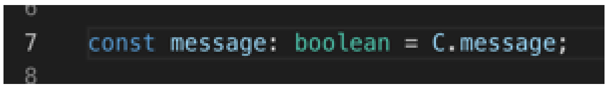
When C is not prepared, the import statement reports a failure.



```
import C from "C";
              Cannot find module 'C'. ts(2307)
              Quick Fix...   Peek Problem
```

Failure to import an “unprepared” package.

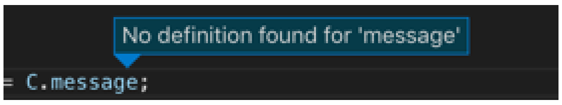
Typing errors are not reported (C.message is a string, not a boolean).



```
const message: boolean = C.message;
```

Failure to report type mistake involving type declared in C.

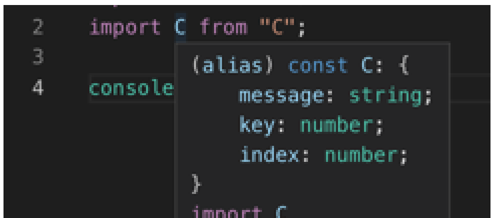
Go to definition does not work.



```
No definition found for 'message'
= C.message;
```

Cannot navigate to variables defined in C.

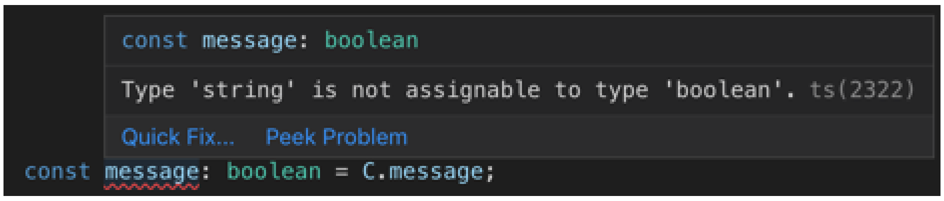
The same tooling works as expected after C is prepared; Imported types are available.



```
import C from "C";
      (alias) const C: {
console     message: string;
            key: number;
            index: number;
        }
        import C
```

Types from C available

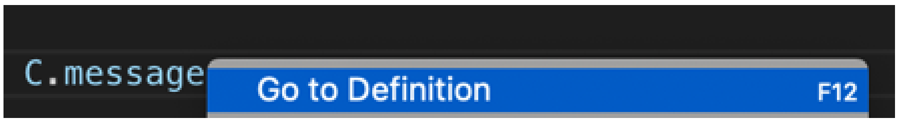
Type mistakes are properly reported.



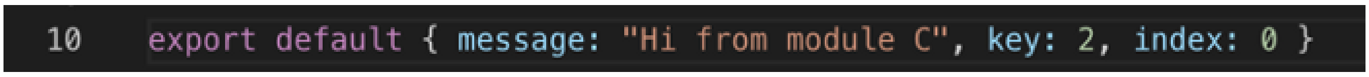
```
const message: boolean
Type 'string' is not assignable to type 'boolean'. ts(2322)
Quick Fix...   Peek Problem
const message: boolean = C.message;
```

Type mistake properly reported

Go-to-definition do bring us to the definition of C.message.



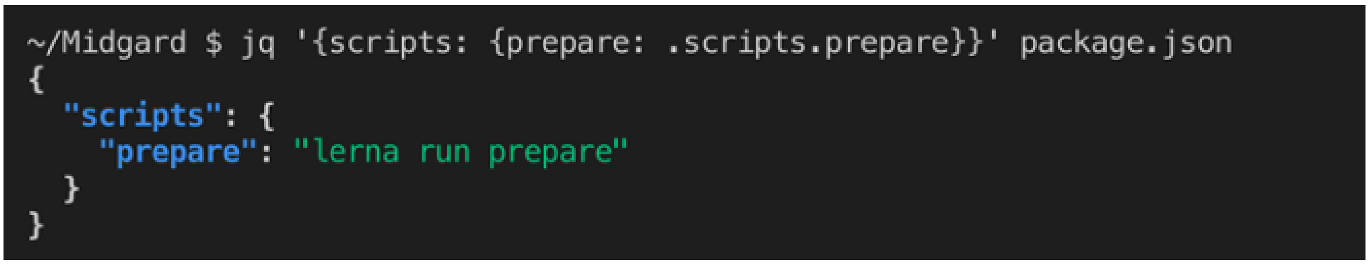
The “Go to Definition” feature…



```
export default { message: "Hi from module C", key: 2, index: 0 }
```

… brings us where we expect

Hopefully this illustrates well why we need to prepare the packages before starting coding if we want to get a decent developer experience. Because a package can be prepared only if its dependencies are already prepared, we need to run the prepare script in all packages in topological order. We use Lerna to orchestrate this.



```
~/Midgard $ jq '{scripts: {prepare: .scripts.prepare}}' package.json
{
  "scripts": {
    "prepare": "lerna run prepare"
  }
}
```

Script getting Midgard ready to work with

Preparing all the packages takes around 5 minutes and needs to be done at each time we change git branch or merge latest master.

Not only our current architecture reduces our productivity by forcing us to have this “prepare” step which takes 5 minutes, but it also has the following effects:

- **Confusing watch output**: Because the interface of a package is its JS and DTS files, a process needs to be run to update them at each time we change a TS file. To do this, each package has a process called “watch”. It is a process which is idle most of the time and updates the JS and DTS files when a TS file changes. We need to run this watch process on every package we may edit. Since we don’t know in advance which packages we will end up editing, we usually use Lerna to run the watch process in each package of the dependency tree of the component we work on. This means that all the watch processes run in the same console making the console output confusing, a build error can easily go unnoticed.

- **IntelliSense implicit dependency:** If we don’t run the “watch” process in every package we modify, IntelliSense will provide wrong type information and some features may be broken. This is because IntelliSense relies on the DTS files. This means that IntelliSense depends on a separate process that we need to run manually, which is not obvious.

- **Incomplete refactoring experience:** When we use DTS files as package interface, IntelliSense cannot provide certain refactoring features. IntelliSense does not seem to always be able to find the consumers of a package public interface. This means that features like “Rename symbol” or “Find all references” do not consistently work cross-packages.

. . .

As mentioned at the beginning of this article, let’s explore an idea which could solve the issues we just described. The idea is to change our packages pubic interfaces from JS+DTS files to TS files.

This idea is illustrated in the tsloader branch of the sample repository.

This is usually considered a bad practice because TS files brings some constrains along with them; the consumer of a TS file should inherit the TypeScript version and TypeScript configuration of its dependencies. We don’t want to add these constrains on repositories consuming Midgard-hosted package, so we will keep publishing JS+DTS files for external consumption. The TS interface will be available only when the consumer is a Midgard-hosted package.

When we consume directly the TS files of our dependencies, the TypeScript compilation is done at bundling time. This is a workflow supported in webpack by various plugins and loaders (babel, ts-loader…) and is common practice in many other TypeScript repositories.

This change would address the issues stated above:

- **Faster setup:** After checking-out a branch or merging latest master, we don't need to do any TypeScript compilation before being able to properly edit the code. This would shave off 5 minutes of the setup time and we believe this is a game changer for Midgard.
- **Clear watch output:** When using a test-app, we don't need to run TSC in watch mode. The only process that is needed is Webpack. Webpack would handle the bundling as well as the type-checking and the linting and order the various outputs in a clear fashion.
- **No implicit dependencies:** IntelliSense will work out of the box as soon as we are done installing our dependencies. There would be no implicit dependency to another process manually launched in parallel.
- **Complete refactoring tooling:** The refactoring tooling is aware of the full dependency graph, so all the features of IntelliSense will work as expected across packages.

While we are very excited about all the benefits we would get from this change, there are still some unknown consequences on Midgard.

Our biggest concern is the performance of IntelliSense. Currently IntelliSense relies on the DTS files to power its features. With the proposed change, no DTS files exist anymore in a development environment so the information about types of other packages must be generated real-time when needed. This is more work for IntelliSense and may slow it down by more than a few seconds. If this is the case, the IntelliSense experience will be perceived as broken and we need to solve this problem before moving ahead with the proposed change.

Another unknown is the scope and cost of this refactoring. This change would introduce some complexity and complexity always has a cost. We don't know what this cost will be because we don't know how coupled our current tooling is with the current architecture.

To mitigate the risks created by these unknowns, we will convert one package at the time starting from the top of the dependency tree. This way, we won't have to solve all the problems at once. While doing so, we will keep a close eye on the performance of IntelliSense.

We will keep gathering feedback, learning from others and communicating on our progress.

JavaScript Typescript Monorepo Webpack