

ANGULAR UNIVERSITY



PREMIUM QUALITY
ANGULAR TUTORIALS

angular-university.io

Typescript 2 Type Definitions Crash Course - Types and Npm, how are they linked ? @types, Compiler Opt-In Types: When To Use Each and Why ?

Typescript has been evolving quickly in the last few months, and one of the things that has evolved more is its most differentiating feature: the Typescript type definitions.

If you have been using Typescript 2 with Angular recently or without it, you might have run into a couple of the following questions or errors situations:

- Does Typescript type safety necessarily mean more ceremony when writing code ?
- What are the multiple types of Typescript Type Definitions ?
- How do I use libraries that don't have Type Definitions available ?
- What has changed in handling node modules from 2.0 to 2.1 ?
- What is the relation between Type Definitions and Npm ?
- When to install third party types ?
- How can packages provide their own custom types ?
- What is `@types`, when should I use it and why ?
- What happened to the `typings` executable and DefinitelyTyped ?

- What are compiler opt-in types, when should I use them and why ?
- Why do I sometimes get a 'duplicate type definition' error, and how to fix it?
- Why does it look like Promise type definitions sometimes don't work correctly ?
- Conclusions and Recommendations on to use Typescript type definitions effectively

We are going to be covering all of this in this post, I invite you to code along (from an empty folder) to get the most out of it, as we will be going over the multiple concepts in an example-driven way.

This is going to be a bit of a bumpy ride, but please have a look at the conclusions section, where we will summarize and give some general guidelines on how to make the most of the available type definitions.

The benefits are very much worth it but it does take some getting used to knowing where to find the right type definitions and knowing which ones to use at any given moment.

OK, I hope you will enjoy the post, so let's get started!

What are the multiple scenarios for Typescript Type Definitions ?

In Typescript 2 and beyond, when using a Javascript library there are now essentially 4 scenarios in what concerns type definitions:

- No type definitions of any kind are available

- Type definitions are available and shipped together with the compiler itself
- A library does not ship with type definitions, but they can be installed separately
- A library ships with its own type definitions built-in

So what is the difference ? Let's start at the beginning: what if there are no type definitions at all ? Because that is a very common case, and will be so for years to come (if not forever).

So let's start with that: we have no guarantee that Javascript modules in the future will be systematically shipped with their own types, or that someone will write those types, publish and maintain them.

The larger and most used modules will likely have good type definitions, but what about smaller modules?

How do I use libraries that don't have Type Definitions available ?

Let's start with a simple example, let's setup a node project in an empty folder and install a simple module named uuid, that generates unique identifiers.

```
1  npm init
2  .... hit enter to all questions
3
4  # install Typescript locally, inside node_modules
5  npm install --save-dev typescript
6
7  # setup a Typescript compiler configuration file tsconfig.json
8  ./node_modules/.bin/tsc --init
9
```

```
10 # install uuid
11 npm install --save uuid
12
```

Please notice that here we have created an initial `package.json` with `npm init`, and have installed a local version of Typescript. If you open this folder with an IDE like for example Webstorm, the Typescript version inside `node_modules` will be taken and used automatically.

So you don't have to install Typescript globally, and its probably better to avoid if possible to avoid version confusions between projects, command line and IDE.

So now that we have installed `uuid`, how do we use it ?

Before and After Typescript 2.1

First, lets check which version of Typescript we are using. Let's say that we are using some version before 2.1, like for example 2.0.10. What happens if we try to import the `uuid` library ? Let's give it a try:

```
1 import * as uuid from 'uuid';
2
3 console.log(uuid());
4
```

OK, so what is going on in that import statement ? Let's break it down:

- we are using the ES6 import syntax to import something from an ES6 module named `uuid`
- we are saying that the module will have a default export because we are using `*`

- We are assigning whatever that single export is and assigning it to a constant named `uuid`, but what type will it have ?

And then we are using the `uuid`, which has implicitly taken the type `any` and we are using it to call it as a function. Let's try to run this to see what happens.

An easy way to run Typescript files

But this is a Typescript file, so we can't call node on it and run it. Do we need a complex build system for that ? No, we could simply create an npm script task that calls the `tsc` compiler and then runs the output using node.

But let's keep the file system clean of generated files, let's instead using an utility called ts-node:

```
1  ### install ts-node
2  npm install --save-dev ts-node
3
```

With `ts-node` installed, let's add a npm script task to run our test program above, which will be in a file called `test.ts`:

```
1  {
2    "name": "types-examples",
3    "scripts": {
4      "demo": "./node_modules/.bin/ts-node ./test.ts"
5    },
6    "devDependencies": {
7      "ts-node": "^2.0.0",
8      "typescript": "^2.1.0"
9    },
10   "dependencies": {
```

```
11     "uuid": "~3.0.1"  
12   }  
13 }
```

OK, so now we can run the test program using a simple npm command:

```
1  
2 npm run demo  
3
```

What will the results be of such a simple program ? This will depend on the Typescript version that you are using, let's see why.

Results Before Typescript 2.1

If you are using for example Typescript 2.0.10, you will get the following error message:

```
test.ts(3,23): error TS2307: Cannot find module 'uuid'.
```

So how can it be that the compiler does not find the module ? We know that the CommonJs `uuid module` is there available in node modules, so why does the compiler say that it can't find it ?

Since 2.0 the compiler should look for modules inside `node_modules` by default, right ?

What is going on here is that there is a `uuid` module present, but `uuid` is not shipped with its own type definitions. So how can use uuid with this version of Typescript ?

Missing Node Type Definitions

One way is to try to use the CommonJs require syntax, so let's try that:

```
1
2  const uuid = require('uuid');
3
4  console.log(uuid());
5
```

We will now receive the following error in the console:

```
test.ts(3,14): error TS2304: Cannot find name 'require'.
```

This is because the function `require` is not known to the Typescript compiler, so it throws an error. So how do we solve this ?

Typescript 2.1 and plain Javascript modules

We will show that further down in this post (how to use Node require in Typescript programs), but right now the simplest solution is to make sure that we are using Typescript 2.1 and above, and use again the import syntax:

```
1  import * as uuid from 'uuid';
2
3  console.log(uuid());
4
```

Now everything is working as expected. With Typescript 2.1, if we have a CommonJs module available inside `node_modules` that has no type definitions available, we will still be able to import it and use it.

But how can we use it as a function, what type is `uuid` then ?

What happens is that this module is being imported and implicitly assigned to the Any Type.

How does the Any Type work ?

The Any type allows up to essentially bypass the type-safety of the Typescript type system:

- we can use Any as a function and call it using parentheses, like we did with `uuid`
- a variable of type Any is assumed to potentially have any property, like a plain Javascript object
- we can also take a variable of Type Any and assign it to essentially anything else (without getting an error)

What does the use of the Type Any mean ?

This means that although our program compiles we are essentially back to writing plain Javascript with that library: we won't have reliable auto-completion or refactoring.

But this also means that any module in npm is available for being seamlessly used in a Typescript program !

So this is a great start. If anything else fails we simply write Javascript and it just works. But how can we improve this and get type safety for the most commonly used npm libraries?

What is the relation between Type Definitions and Npm ?

Let's now cover modules that ship with their own types. There are more and more modules each day that get shipped in npm with their own type-definitions already built-in.

This means that the types are shipped directly inside the node module itself, and don't have to be installed separately.

Let's start with a example, let's for example install the Axios isomorphic HTTP library. If you don't know Axios, its a great library for doing both Ajax on the client and HTTP calls on the server, while using the same Promise-based API:

```
1
2  npm install --save axios
3
```

This command will install Axios, so we could start using this client to query a REST API using plain Javascript. The good news is: we can also do the same thing in a type-safe way as well !

This is because Axios comes with its own type definitions built-in. Let's have a look:

```
1
2  cd node_modules/axios
3
4  bash-3.2$ ls -1 *.d.ts
5
6  index.d.ts
7
```

As we can see, the Axios node module comes bundled with its own Type definitions, so we don't need to install them separately ! So let's see this

in action in our program:

```
1
2 import axios from 'axios';
3 import {AxiosPromise} from "axios";
4
5 const response: AxiosPromise = axios.get('/lessons', {
6     ...
7 });
8
```

The Axios library has a default export that we have imported and gave the name `axios`. This import has implicitly the type `AxiosStatic` as declared on the Axios type definition file.

Do we really need type annotations to get type-safety ?

This import named `axios` is not of type `Any`, and we have auto-completion available as well as refactoring and find usages, all working out of the box.

More than that, do you see the `AxiosPromise` type annotation ? Its actually redundant, if we delete it the type of the response the constant would still be inferred as being of type `AxiosPromise`, and we would have auto-completion working for that variable as well.

More than that, do you see the configuration object after the url ? That is automatically inferred to be of type `AxiosRequestConfig`, so we have auto-completion to fill in the request parameters as well.

Type safety does not mean more ceremony when coding

So why don't we get a compilation error at this stage because the object is empty ?

That is because the `AxiosRequestConfig` type definition only has optional properties. With our IDE we can jump into the definition of `AxiosRequestConfig`:

```
1
2  export interface AxiosRequestConfig {
3      url?: string;
4      method?: string;
5      baseURL?: string;
6      ....
7  }
8
```

As we can see, all the properties are marked as optional using a question mark. So this is a good example of how using a library with built-in type definitions does not mean more verbosity in our program, or getting constant compiler errors.

The biggest advantage of Typescript

With Typescript, we can mostly have both the convenience of plain Javascript plus the enhanced tooling. If we use libraries that provide their own built-in types, we can have auto-completion, refactoring and find usages almost everywhere in our program, at the expense of using just a few type annotations at strategic places.

The biggest exception for this will be function parameters, where there is no way for the compiler to infer what is the type of a function parameter. But its a great idea to mention the types of our function parameters for documentation purposes.

How to make the most of Typescript type definitions

If you want to leverage Typescript type inference to its maximum and have it auto-detect the type of the largest amount possible of variables, the best way is to go to the `tsconfig.json` and set the `noImplicitAny` property to true:

```
1  {
2    "compilerOptions": {
3      "module": "commonjs",
4      "target": "es5",
5      "noImplicitAny": true,
6      "sourceMap": false
7    }
8  }
```

This way, if by some reason the compiler can't infer the type of a variable, it will not implicitly assign it the type any. This is probably one of the most important properties available to configure the compiler.

We could almost have named it `useTypeInferenceAsMuchAsPossible` instead of `noImplicitAny`

Now let's take the Axios type definitions, and see what we can do with them in terms of helping us to build a type-safe program.

Leveraging the Promise API in our programs

We would like to be able to define a method that does an HTTP call and returns a Promise of a given type:

```

1
2  import axios from 'axios';
3  import {AxiosPromise} from "axios";
4
5
6  interface Lesson {
7      id:number;
8      description: string;
9  }
10
11
12  function getLesson(lessonId:number): AxiosPromise<Lesson> {
13      return axios.get(`lessons/${lessonId}`);
14  }
15
16
17  const promise = getLesson(1);
18
19
20  promise.then(response => {
21      ...
22  });
23
24
25

```

So what have we done in this small program ? Let's break it down:

- we have created a custom object type called `Lesson`, with two mandatory properties
- we have defined a function `getLesson` that does an HTTP call and returns a Promise
- We are trying to specify what is the data returned by the promise via a generic parameter in the `AxiosResponse` type

So our goal here is to have type safety in the `then` clause, by knowing there implicitly that the data returned is `Lesson`, without having to use a

type annotation.

Trying to use generics in promise return types

So what is the result ? We currently get two compiler errors:

```
Error:(13, 38) TS2315: Type 'AxiosPromise' is not generic.  
Error:(21, 14) TS7006: Parameter 'response' implicitly has  
an 'any' type.
```

So what do these errors mean ? Let's break it down:

- we can't specify the data returned by an Axios promise, via a generic parameter
- this means that the data returned is implicitly of type Any
- the promise response is also implicitly of type any, which throws an error

At this point, we can see that the Promise type definitions shipped with Axios although useful, would need some extra type annotations to ensure type safety on our program. For example:

```
1  function getLesson(lessonId:number) {  
2      return axios.get(`lessons/${lessonId}`);  
3  }  
4  
5  const promise = getLesson(1);  
6  
7  promise.then(response => {  
8  
9      const lesson: Lesson = response.data;  
10  
11      ...  
12  }
```

```
13  });  
14
```

And this would work great, have a look at line 9 we added an explicit type annotation.

But at this point we could think that Promise is a standard ES6 API, so why not use that instead ? We would benefit both from a standard API and type inference in our program.

So if this would be a node program, how could we do that ?

Writing Node programs using the standard Promise API

What we need to do is to use another library that does not necessarily ship with its own promise type definitions, or in this case that returns Promise-like types that are compatible with ES6 promises.

Let's for example set up `request-promise`, which is a promise enabler for the popular `request` node HTTP client:

```
1  
2  npm install --save request  
3  npm install --save request-promise  
4
```

So how can we use this client to write type-safe node programs, using the standard promise API ?

Using Node require in Typescript programs

The way that `request-promise` works is that it has the same API as `request`, but it returns promises. So how can we use it ? We could start by using it as a plain node module, by requiring it like this:

```
1  
2  const rp = require('request-promise');  
3
```

But at this point we would get an error:

```
Error:(3, 12) TS2304:Cannot find name 'require'.
```

What is happening here is that we have no type definition for this global function named `require`.

The Node runtime does not ship with its own type definitions, so we need to import those types separately. Where can we find them ? They are also in npm but need to be installed separately.

We can install the node runtime type definitions in the following way:

```
1  
2  npm install @types/node --save-dev  
3
```

So what did this do, what is this `@types` module ?

What is `@types`, when should I use it and why ?

This `@types` scoped package is where we can find a ton of useful type definitions, such as for example the type definitions of node that allow us to use `require` for example.

But if you have been following Typescript for a while, you might remember something called DefinitelyTyped, and a typings executable, that we used to use before to install type definitions.

What happened to the `typings` executable and DefinitelyTyped ?

If we head over to the npm page of the `@types` scoped package, we can see what happened there - [@types](https://www.npmjs.com/@types).

As we can see, all the content of DefinitelyTyped is now available under the `@types` scoped package, and we don't need anymore to use the typings executable to download type definitions.

We can now simply use npm and the Typescript compiler will implicitly take any type definitions installed inside the `node_modules/@types` folder and include it during compilation transparently.

When to use @types then ?

The `@types` scope package contains type definitions for a lot of libraries, like Express, Sequelize, JQuery, and many others. So definitely have a look there if you are missing some type definitions, but make sure of two things first:

- check if the package you are using already has types built-in, and if so prefer those

- check if type definitions are already shipped with the compiler, more on this later

The type definitions inside `@types` are super helpful, but some of those types might not be appropriate anymore in certain situations. Let's give an example with the Promises library.

Using Request Promise to build a type safe promise call

Let's start by installing the type definitions for `request-promise` available in `@types`, and if you have been coding along its a great time to uninstall axios to avoid library conflicts:

```
1
2  npm uninstall axios --save
3
4  npm install --save-dev @types/request-promise
5
```

Now that we have the type definitions for `request-promise`, this is the program that we would like to be able to write:

```
1  import * as rp from 'request-promise';
2
3
4  interface Lesson {
5      id: number;
6      description: string;
7  }
8
9
10 function getLesson(lessonId: number): Promise<Lesson> {
11     return rp.get(`lessons/${lessonId}`);
12 }
```

```
13
14  const promise = getLesson(1);
15
16  promise.then(lesson => {
17      .... we want this lesson variable to be implicitly of type Lesson
18  });
```

But at this stage, we get an error:

```
Error:(11, 38) TS2304: Cannot find name 'Promise'.
```

Understanding the Cannot Find Promise common issue

So it looks like the node runtime type definitions that we added to our program don't include promises. So let's have a look at `@types` to see where we can find them.

Please do read the conclusions section on this, but right now let's say that we would bring some type definitions from `@types` for Promises:

```
1
2  npm install --save-dev @types/es6-promise
3
```

So what will this do ? IT will install type definitions for ES6 promises, which includes a generic parameter for the Promise type.

So now we can say that this function returns a Promise of Lesson, and have the type of the variable lesson to be correctly inferred by the compiler as being of type Lesson.

Which is exactly what we wanted, but there is a huge catch. And its a good example of how we should not take all the types available in `@types` systematically, but pick and choose.

What is the catch with the use of es6-promise ?

To understand what the problem is, let's try the following program, which should throw an error:

```
1  import * as rp from 'request-promise';
2
3  interface Lesson {
4      id:number;
5      description: string;
6  }
7
8
9  function getLesson(lessonId:number): Promise<Lesson> {
10      return rp.get(`lessons/${lessonId}`)
11          .then((lesson:any) => Promise.resolve(lesson.description));
12  }
```

Can you see the issue ? The function is returning a Promise of string, because the lesson has been transformed into a string by the then clause.

But yet the program compiles without any error. So what is going on here ?

Not all type definitions leverage completely the Typescript type system

As we have seen before, not all type definitions leverage the type system to its maximum extent. This is also because the Typescript compiler

features keep evolving so fast that not all type definitions leverage all the latest features.

Which can be great because we might not want to use generics all the time in our programs, so returning Any in our API and assuming the caller will add a type annotation is a viable solution as well.

But in this case we would really like to use Promise with a generic parameter, because its a really great fit for that. It really makes sense to specify in our program what type of data is the Promise expected to return, and use that information to type check the program.

So what can we do ? It turns out that the Typescript compiler itself ships with a ton of Type definitions ready to use, and one of them is Promises.

What are compiler opt-in types, when should I use them and why ?

Have a look at the compiler options, at the `--lib` flag [here](#).

There are a ton of type definitions available that come bundled with the compiler, including for example all the type definitions of ES6 itself, which includes Promises.

So we can simply leverage these by using the `lib` compiler flag:

```
1  {
2      "compilerOptions": {
3          "module": "commonjs",
4          "target": "es5",
5          "noImplicitAny": true,
6          "sourceMap": false,
7          "lib": ["es6"]
8      }
9  }
```

```
8     }  
9 }
```

So with this, the compiler knows of a Promise type that is built-in with the compiler. So if we compile our program with that turned on, what do we get ?

Initially we will get a few errors:

```
node_modules/@types/es6-promise/index.d.ts(42,19): error  
TS2300: Duplicate identifier 'Promise'.  
node_modules/typescript/lib/lib.es2015.iterable.d.ts(145,1  
1): error TS2300: Duplicate identifier 'Promise'.  
...
```

Why do I sometimes get this 'duplicate type definition' error ?

In this case, we have a type definition named Promise in two places:

- one via `@types/es6-promise`
- the other via the built-in compiler types that we have opted-in

So how do we solve this duplicate type definition issue ? The solution is to uninstall the types that we had installed via `@types` for promises:

```
1  
2 npm uninstall --save-dev @types/es6-promise  
3
```

Now we are only going to get this errors:

```
test.ts(11,12): error TS2322: Type 'Bluebird<any>' is not
assignable to type 'Promise<Lesson>'.
Property '[Symbol.toStringTag]' is missing in type
'Bluebird<any>'.
```

Do make sure to have a look at the conclusions to summarize what is going on here.

It turns out that `@types/request-promise` already comes with its own Promise type definitions after all: Because those type definitions use internally Bluebird type definitions (Bluebird is a great promise library, used by Sequelize for example - a Node ORM).

So what can we do at this point ? Because it looks like the type definitions of `@types/request-promise` are incompatible with the types from the ES6 built-in type definitions at the moment.

This a temporary situation because Bluebird promises used to be compatible with ES6 promises for a long time, and actually this is likely to have already been solved by the time you read this post.

What to learn from this example ?

This is a good example on how using the latest type definitions might not always be viable or the best approach at any given time. Its an option, but needs to be weighted against other things.

Sometimes using the simpler types that are available like `AxiosPromise` or the return of `request-promise` calls is a much more viable

alternative, and then simply add a few extra type annotations where necessary, in case those APIs return Any.

its not because an API returns an explicit any that we need to use any also in our own program

So as general guideline, its better to choose packages that come with built-in types and use those types as much as possible via type inference, only importing third party types if needed and picking and choosing the best fit at each moment.

What would the error look like that we where looking for ?

Let's try to see if Typescript catches the error that we are trying it to throw with a simpler example (taken from this [issue](#) report). How about this:

```
1
2 interface Foo {
3     a: string
4     b: number
5 }
6
7 class Hello {
8     getFoo(): Promise<Foo> {
9         return Promise.resolve("a string")
10    }
11 }
12
13
```

This would throw as expected the following error:

```
test.ts(31,16): error TS2322: Type 'Promise<string>' is
not assignable to type 'Promise<Foo>'.
  Type 'string' is not assignable to type 'Foo'.
```

So the compiler is prepared to handle the detection of this error. Its just that the library type definitions available need to evolve over time to leverage this functionality, and there will likely always be some sort of gap.

Handling the gap between libraries and the compiler

The Typescript compiler will apply the latest type checks to any type definitions available in node modules, including `@types`.

To avoid this, and ensure that only our program is checked by the compiler we can use the flag `skipLibCheck` to true.

This prevents the more recent versions of the compiler to throw errors against ancient libraries, and also has the added benefit of increasing the performance of your build in a noticeable way.

OK so now we have reached the end, it was a bit of a roller coaster but this is very close to the every day issues that you will find while developing with Typescript.

Let's try to make sense of it all, and try to come with some general guidelines to use the type system effectively in the next section.

Conclusions and Recommendations

If we are using Typescript, we have multiple sources of Type Definitions, and knowing which ones to choose and why is essential to be able to

have a good developer experience.

We need to be aware of one thing: the compiler is always adding more features, but the libraries available might not leverage them yet.

When should we use compiler opt-in types?

It's a great idea to use the compiler built-in types (the `lib` flag) as much as possible, because those types are written to maximize the type safety of our programs, and make sure we conform to standard APIs.

But depending on the current state of existing libraries, it might introduce other issues. When opting-in to compiler built-in types, do make sure that they don't conflict with types that we already imported before like `es6-promise`, and that they don't affect third party types if not needed by using `skipLibCheck`.

But don't feel obliged to use opt-in types, there is a good reason why right now they are turned off by default, it's because of backwards compatibility with part of the existing ecosystem.

When should we use `@types` ?

It's better to instead of using `@types` systematically, to try to use the built-in types of each module as much as possible, and use `@types` strategically if necessary for example for modules like Express or Sequelize.

Plain Javascript modules like those two will likely make the bulk of your program and have great types available on `@types`. But for example newer modules like Firebase: they already come with types now.

So if you also install `@types/firebase` you will run into duplicate type issues.

On the other hand things like `@types/node` are essential to write any node program.

So the suggestion here is: have a look at the built-in types to see if there is anything there similar to what you are looking for. Have a look at the node module itself to see if it has already types inside it: this will be more and more common.

If no types are found neither inside the module nor built-in to the compiler, then have a look at `@types` to see if there are some good types there. Fast forwarding to the future, the ideal would be that `@types` no longer exists, and that most libraries ship their own type definitions.

But that won't happen anytime soon and it's great to have those high quality types there available.

What if no type definitions are available ?

If no type definition files are available for a given library, that will not prevent us from using it. With Typescript 2.1, we can import the library directly and anything imported will be assigned the type `Any`.

So we can seamlessly integrate with any existing Javascript module without special integration needed.

How to make sure our programs leverage type safety effectively ?

One of the best things we can do other than carefully choosing the types we add to our program, is to turn `noImplicitAny` to true.

This will have the effect that the compiler will be able to infer almost all types in your program, at the expense of only a few type annotations, especially in function arguments which is a great thing to add anyway.