Basic Typescript for Angular: Understanding Modules

## **Introduction**

You are starting to learn Angular, but it's hard to know what's Angular and what's Typescript.

This guide gives an understanding of how Typescript modules work and how they are used in Angular. What are these import and export statements I see generated by the Angular CLI? How do I create my own modules?

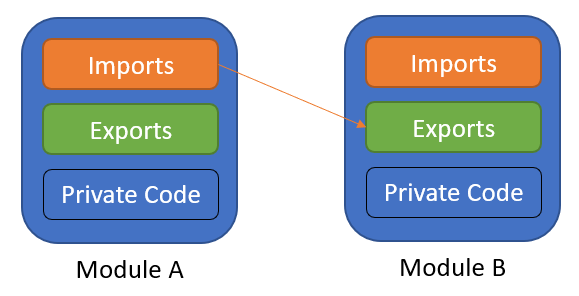
To answer these questions and demonstrate the concepts we need to learn, we will use the scenario of working on shopping cart functionality within an app.

## **What Are Typescript Modules?**

The goal of modular code is that each individual module provides a piece of functionality, exposed through a well-defined interface. The internal details of how a module works are isolated, making it easier to test and refactor.

Modules can make use of other modules by importing them. In turn, a module exports only what it wants other code to be able to use.

In Typescript, a module is simply a file that imports or exports something.



In order for the app to run, the dependencies between modules above are resolved via a module loader. The browser cannot load the code for Module A before Module B has been loaded.

Typescript does not provide a runtime, it's just a transpiler into javascript. The Angular CLI takes care of this module loading aspect for you. When it builds your Angular app, it uses WebPack as the loader technology it is using.

### **Angular Modules vs. Typescript Modules**

Angular also has the concept of modules, which you will see in the code as @Module() definitions. These modules are independent of Typescript modules.

Angular modules follow the same software concept of modularization, but at a different level. They aim to separate the app into functional areas, complete with UI and service definitions.

## **Creating a Module**

Creating a module is as simple as creating a Typescript file that has an import or export statement.

A module can export one or more declarations: a class, function, interface, enum, constant, or type alias.

For this guide's scenario, we'll look at the ProductsService and related types we need for an e-commerce application.

1

2

3

4

5

6

7

8

9

10

11

12

// app/shopping-cart/products.service.ts

export class ProductsService {

// .. Service code here

}

export interface Product {

// Interface declarations

}

// Private function to this module, not on the global namespace

function logDebug(message: string) { console.log(message); }

typescript

From our products.service.ts Typescript module above, only the ProductsService class and Product interface are exported. They are the only types that are available to any importers of this module.

The logDebug function is private, for use only within this module.

With non-modular javascript, the logDebug function would have been placed on the global namespace. This can lead to unexpected consequences if some other loaded javascript overrides or otherwise changes this function.

Recall that all Typescript modules are isolated and so operate on their own scope, not the global scope. The logDebug function is only available within this module. Another module is safe to declare its own function called logDebug and it will in no way conflict with this one.

### **Ways of Exporting**

There are a few ways declarations can be exported from a module.

The typical way it's done in Angular is as we have already seen, immediately exporting when the declaration is made:

1

2

3

4

5

6

7

8

// Export at time of declaration

export class ProductsService {

// .. Service code here

}

export interface Product {

// Interface declarations

}

typescript

Alternatively, you can also export one or more declarations in a single export statement:

1

2

3

4

5

6

7

8

9

10

class ProductsService {

// .. Service code here

}

interface Product {

// Interface declarations

}

// Export as a single statement

export { ProductsService, Product }

typescript

This option keeps all the exports in place, which has the advantage of making it clear to see the module's exported public interface.

## **Using a Module**

To make use of our module, we need to import it.

Let's assume that we have a cart.component.ts that needs to make use of the ProductsService. It can import it like this:

1

2

3

// app/shopping-cart/cart.component.ts

import { ProductsService } from './products.service';

typescript

This is importing the ProductsService class from our module products.service.ts.

We are able to import the ProductsService from our module because it has been exported. If we try to import the logDebug function, we will get an error at compile time:

1

2

3

4

// cart.component.ts

//ERROR: logDebug is not exported, so cannot be imported

import { logDebug } from './products.service';

typescript

### **Module Resolution**

Typescript has a concept of module resolution which it uses at compile-time to find the intended module to import.

In the previous examples, the reference to the module in the import statement is a relative path, so we are expecting the products.service.ts to be a sibling to the cart.component.ts file.

Notice that the .ts extension is not needed! Our Typescript is actually going to be transpiled into javascript, and so the final module has a .js extension.

We could be importing some other file extension too: a .tsx or a .d.ts from an NPM package.

This is module resolution at work. For our Angular apps, it's not something we need to be concerned about, but it is worth knowing that this is why there is no file extension in the import statement.

## **Aliasing Types**

There will be occasions that two modules that we want to use are going to export a type with the same name.

When we try to import Product from another module, say an e-commerce CMS, we will get an error:

1

2

3

4

import { ProductsService, Product } from './products.service';

// ERROR: Duplicate identifier 'Product'

import { Product } from 'ecommerceCMS/products';

typescript

To solve this, we can alias the type so that we avoid the naming clash:

1

2

3

4

import { ProductsService, Product } from './products.service';

// Now available as CMSProduct

import { Product as CMSProduct} from 'ecommerceCMS/products';

typescript

Alternatively, you can import all types from a module into a variable:

1

2

import \* as products from './products.service';

import { Product } from 'ecommerceCMS/products';

typescript

This places all of the types from our module into a products variable. You can reference the types as products.ProductsService and products.Product, avoiding a naming clash with the ECommerce CMS Product.

## **Rollup Exports into Barrels**

As our Shopping Cart functionality grows, we're going to end up with more and more modules. For any importers, this can mean there will be a lot of import statements and it can be difficult to maintain.

We ideally want to roll-up all of our smaller modules into a single one, let’s call it shoppingCart.

This is done through barrels. Barrels are modules which pull lots of individual modules together, reexporting their declarations, so creating a single cohesive module.

### **Defining a Barrel**

To define a barrel for our Shopping Cart, we create a file named index.ts in the shopping cart root folder, that simply re-exports our other modules:

1

2

3

4

// app/shoppingCart/index.ts

export { ProductService, Product } from './shoppingCart/products.service';

export { CartComponent } from './shoppingCart/cart.component'

typescript

### **Importing a Barrel**

A barrel is just like any other module, so we can import it in the same way. However, we defined our barrel in a file named index.ts.

Like a web server serving up a default page, the Typescript Module Resolution process as the same concept with index.ts.

When resolving an import statement, it will check the referenced path. If it is a directory and there is an index.ts file, then the import statement will resolve:

1

2

3

4

5

// Simple import from the barrel

import { ProductService, Product, CartComponent } from './shoppingCart';

// Exactly the same, but references the index module directly

import { ProductService, Product, CartComponent } from './shoppingCart/index';

typescript

### **Safer Refactoring**

We now have a single module, shoppingCart.ts, which is intended to be used within the app.

Using a barrel helps when we start to do any refactoring, which can otherwise lead to a ripple-effect change across the app.

We decide that our ProductService should be in a "services" sub-folder. This means that everywhere the service is imported would need the path updated.

Without using a barrel, this could be a lot of places! However, with the barrel, we just change the one export statement to match:

1

2

3

4

5

// app/shoppingCart/index.ts

// New path to products.service.ts

export { ProductService, Product } from './shoppingCart/services/products.service';

export { CartComponent } from './shoppingCart/cart.component'

typescript

Even if we decide to rename the ProductService to be ProductApiService, we can still export from our barrel a ProductService declaration to enable backward compatibility:

1

2

3

4

5

// app/shoppingCart/index.ts

// ProductService alias to aid backwards compatibility

export { ProductApiService, ProductApiService as ProductService, Product } from './shoppingCart/services/products.service';

export { CartComponent } from './shoppingCart/cart.component'

typescript

## **Conclusion**

You now have an understanding of what Typescript modules are and how they are used when creating Angular Apps.

You know how to take control of the import and export statements to avoid name clashes or to simplify refactoring.

All this knowledge is brought together in the concept of barrels, creating a unified module from other smaller modules. Barrels are a good idea to prevent a ripple-effect change through your app as you refactor code.