Circular dependency

A circular dependency occurs when two classes depend on each other. For example, class A needs class B, and class B also needs class A. Circular dependencies can arise in Nest between modules and between providers.

While circular dependencies should be avoided where possible, you can't always do so. In such cases, Nest enables resolving circular dependencies between providers in two ways. In this chapter, we describe using **forward referencing** as one technique, and using the **ModuleRef** class to retrieve a provider instance from the DI container as another.

We also describe resolving circular dependencies between modules.

> warning **Warning** A circular dependency might also be caused when using "barrel files"/index.ts files to group imports. Barrel files should be omitted when it comes to module/provider classes. For example, barrel files should not be used when importing files within the same directory as the barrel file, i.e. `cats/cats.controller` should not import `cats` to import the `cats/cats.service` file. For more details please also see this github issue.

Forward reference

A **forward reference** allows Nest to reference classes which aren't yet defined using the `forwardRef()` utility function. For example, if `CatsService` and `CommonService` depend on each other, both sides of the relationship can use `@Inject()` and the `forwardRef()` utility to resolve the circular dependency. Otherwise Nest won't instantiate them because all of the essential metadata won't be available. Here's an example:

```
@@filename(cats.service)
@Injectable()
export class CatsService {
  constructor(
    @Inject(forwardRef(() => CommonService))
    private commonService: CommonService,
  ) {}
}
@@switch
@Injectable()
@Dependencies(forwardRef(() => CommonService))
export class CatsService {
  constructor(commonService) {
    this.commonService = commonService;
  }
}
```

> info **Hint** The `forwardRef()` function is imported from the `@nestjs/common` package.

That covers one side of the relationship. Now let's do the same with `CommonService`:

```
@@filename(common.service)
@Injectable()
export class CommonService {
  constructor(
    @Inject(forwardRef(() => CatsService))
    private catsService: CatsService,
  ) {}
}
@@switch
@Injectable()
@Dependencies(forwardRef(() => CatsService))
export class CommonService {
  constructor(catsService) {
    this.catsService = catsService;
  }
}
```

> warning **Warning** The order of instantiation is indeterminate. Make sure your code does not depend on which constructor is called first. Having circular dependencies depend on providers with `Scope.REQUEST` can lead to undefined dependencies. More information available here

ModuleRef class alternative

An alternative to using `forwardRef()` is to refactor your code and use the `ModuleRef` class to retrieve a provider on one side of the (otherwise) circular relationship. Learn more about the `ModuleRef` utility class here.

Module forward reference

In order to resolve circular dependencies between modules, use the same `forwardRef()` utility function on both sides of the modules association. For example:

```
@@filename(common.module)
@Module({
  imports: [forwardRef(() => CatsModule)],
})
export class CommonModule {}
```

That covers one side of the relationship. Now let's do the same with `CatsModule`:

```
@@filename(cats.module)
@Module({
  imports: [forwardRef(() => CommonModule)],
})
export class CatsModule {}
```