nest

Custom providers

There are a lot of scenarios when you might want to bind something **directly** to the Nest inversion of control container. For example, any constant values, configuration objects created based on the current environment, external libraries, or pre-calculated values that depends on few other defined providers. Moreover, you are able to override default implementations, e.g. use different classes or make use of various test doubles (for testing purposes) when needed.

One essential thing that you should always keep in mind is that Nest uses **tokens** to identify dependencies. Usually, the auto-generated tokens are equal to classes. If you want to create a custom provider, you'd need to choose a token. Mostly, the custom tokens are represented by either plain strings or symbols. Following best practices, you should hold those tokens in the separated file, for example, inside `constants.ts`.

Let's go through the available options.

Use value

The `useValue` syntax is useful when it comes to either define a constant value, put external library into Nest container, or replace a real implementation with the mock object.

```
import { connection } from './connection';

const connectionProvider = {
  provide: 'CONNECTION',
  useValue: connection,
};

@Module({
  providers: [connectionProvider],
})
export class ApplicationModule {}
```

In order to inject a custom provider, we use the `@Inject()` decorator. This decorator takes a single argument - the token.

JS

```
@Injectable()
export class CatsRepository {
  constructor(@Inject('CONNECTION') connection: Connection) {}
}
```

HINT
The `@Inject()` decorator is imported from `@nestjs/common` package.

When you want to override a default provider's value, let's say, you'd like to force Nest to use a mock `CatsService` due to testing purposes, you can simply use an existing class as a token.

```
import { CatsService } from './cats.service';

const mockCatsService = {};
const catsServiceProvider = {
  provide: CatsService,
  useValue: mockCatsService,
};

@Module({
  imports: [CatsModule],
  providers: [catsServiceProvider],
})
export class ApplicationModule {}
```

In above example, the `CatsService` will be overridden by a passed `mockCatsService` mock object. It means, that Nest instead of creating `CatsService` instance manually, will treat this provider as resolved already, and use `mockCatsService` as its representative value.

Use class

The `useClass` syntax allows you using different class per chosen factors. For example, we have an abstract (or default) `ConfigService` class. Depending on the current environment, Nest should use a different implementation of the configuration service.

```
const configServiceProvider = {
  provide: ConfigService,
  useClass:
    process.env.NODE_ENV === 'development'
      ? DevelopmentConfigService
      : ProductionConfigService,
};

@Module({
  providers: [configServiceProvider],
})
```

```
export class ApplicationModule {}
```

NOTICE
Instead of a custom token, we have used the `ConfigService` class, and therefore we have overridden the default implementation.

In this case, even if any class depends on `ConfigService`, Nest will inject an instance of the provided class (`DevelopmentConfigService` or `ProductionConfigService`) instead.

Use factory

The `useFactory` is a way of creating providers **dynamically**. The actual provider will be equal to a returned value of the factory function. The factory function can either depend on several different providers or stay completely independent. It means that factory may accept arguments, that Nest will resolve and pass during the instantiation process. Additionally, this function can return value **asynchronously**. It's explained in more detail here. Use it when your provider has to be dynamically calculated or in case to resolve an asynchronous operation.

JS

```
const connectionFactory = {
  provide: 'CONNECTION',
  useFactory: (optionsProvider: OptionsProvider) => {
    const options = optionsProvider.get();
    return new DatabaseConnection(options);
  },
  inject: [OptionsProvider],
};

@Module({
  providers: [connectionFactory],
})
export class ApplicationModule {}
```

HINT
If your factory needs other providers, you have to pass their tokens inside the `inject` array. Nest will pass instances as arguments of the function in the same order.

Use existing

The `useExisting` allows you creating aliases for existing providers. For example, the token `AliasedLoggerService` is an alias for `LoggerService`.

```
@Injectable()
class LoggerService {}

const loggerAliasProvider = {
  provide: 'AliasedLoggerService',
  useExisting: LoggerService
};

@Module({
  providers: [LoggerService, loggerAliasProvider],
})
export class ApplicationModule {}
```

HINT
The instance of `LoggerService` will be equal to the instance defined by `AliasedLoggerService` token.

Export custom provider

In order to export a custom provider, we can either use a token or a whole object. The following example shows a token case:

JS

```
const connectionFactory = {
  provide: 'CONNECTION',
  useFactory: (optionsProvider: OptionsProvider) => {
    const options = optionsProvider.get();
    return new DatabaseConnection(options);
  },
  inject: [OptionsProvider],
};

@Module({
  providers: [connectionFactory],
  exports: ['CONNECTION'],
})
export class ApplicationModule {}
```

But you can use a whole object as well:

JS

```js
const connectionFactory = {
  provide: 'CONNECTION',
  useFactory: (optionsProvider: OptionsProvider) => {
    const options = optionsProvider.get();
    return new DatabaseConnection(options);
  },
  inject: [OptionsProvider],
};

@Module({
  providers: [connectionFactory],
  exports: [connectionFactory],
})
export class ApplicationModule {}
```

Support us

Nest is an MIT-licensed open source project. It can grow thanks to the support by these awesome people. If you'd like to join them, please read more here.

Principal Sponsor

Valor SOFTWARE

Sponsors / Partners

Become a sponsor