Logger

Nest comes with a default implementation of internal `Logger` that is used during the instantiation process and also, in several different situations, such as **occurred exception**, and so on. But sometimes, you might want to disable logging entirely, or provide a custom implementation and handle messages on your own. In order to turn off a logger, we use a Nest's options object.

```
const app = await NestFactory.create(ApplicationModule, {
  logger: false,
});
await app.listen(3000);
```

Nevertheless, we could want to use a different logger under the hood, instead of disabling a whole logging mechanism. In order to do that, we have to pass an object that fulfills `LoggerService` interface. An example could be a built-in `console`.

```
const app = await NestFactory.create(ApplicationModule, {
  logger: console,
});
await app.listen(3000);
```

But it's not an apt idea. However, we can create our own logger easily.

```
import { LoggerService } from '@nestjs/common';

export class MyLogger implements LoggerService {
  log(message: string) {}
  error(message: string, trace: string) {}
  warn(message: string) {}
  debug(message: string) {}
  verbose(message: string) {}
}
```

Then, we can apply `MyLogger` instance directly:

```
const app = await NestFactory.create(ApplicationModule, {
  logger: new MyLogger(),
});
await app.listen(3000);
```

Extend built-in logger

Lot of use cases require creating your own logger. You don't have to entirely reinvent the wheel though. Simply extend built-in `Logger` class to partially override the default implementation, and use `super` to delegate the call to the parent class.

```
import { Logger } from '@nestjs/common';

export class MyLogger extends Logger {
  error(message: string, trace: string) {
    // add your tailored logic here
    super.error(message, trace);
  }
}
```

Dependency injection

If you want to enable dependency injection in your logger, you have to make the `MyLogger` class a part of the real application. For instance, you can create a `LoggerModule` .

```
import { Module } from '@nestjs/common';
import { MyLogger } from './my-logger.service.ts';

@Module({
  providers: [MyLogger],
  exports: [MyLogger],
})
export class LoggerModule {}
```

Once `LoggerModule` is imported anywhere, the framework will take charge of creating an instance of your logger. Now, to use the same instance of a logger across the whole app, including bootstrapping and error handling stuff, use following construction:

```
const app = await NestFactory.create(ApplicationModule, {
  logger: false,
```

```
});
app.useLogger(app.get(MyLogger));
await app.listen(3000);
```

The only downside of this solution is that your first initialization messages won't be handled by your logger instance, though, it shouldn't really matter at this point.

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

Sponsors / Partners

Become a sponsor