Lifecycle Events

A Nest application, as well as every application element, has a lifecycle managed by Nest. Nest provides **lifecycle hooks** that give visibility into key lifecycle events, and the ability to act (run registered code on your modules, providers or controllers) when they occur.

Lifecycle sequence

The following diagram depicts the sequence of key application lifecycle events, from the time the application is bootstrapped until the node process exits. We can divide the overall lifecycle into three phases: **initializing**, **running** and **terminating**. Using this lifecycle, you can plan for appropriate initialization of modules and services, manage active connections, and gracefully shutdown your application when it receives a termination signal.

Lifecycle events

Lifecycle events happen during application bootstrapping and shutdown. Nest calls registered lifecycle hook methods on modules, providers and controllers at each of the following lifecycle events (**shutdown hooks** need to be enabled first, as described below). As shown in the diagram above, Nest also calls the appropriate underlying methods to begin listening for connections, and to stop listening for connections.

In the following table, `onModuleDestroy`, `beforeApplicationShutdown` and `onApplicationShutdown` are only triggered if you explicitly call `app.close()` or if the process receives a special system signal (such as SIGTERM) and you have correctly called `enableShutdownHooks` at application bootstrap (see below **Application shutdown** part).

Lifecycle hook method	Lifecycle event triggering the hook method call
`onModuleInit()`	Called once the host module's dependencies have been resolved.
`onApplicationBootstrap()`	Called once all modules have been initialized, but before listening for connections.
`onModuleDestroy()*`	Called after a termination signal (e.g., `SIGTERM`) has been received.
`beforeApplicationShutdown()*`	Called after all `onModuleDestroy()` handlers have completed (Promises resolved or rejected); once complete (Promises resolved or rejected), all existing connections will be closed (`app.close()` called).
`onApplicationShutdown()*`	Called after connections close (`app.close()` resolves).

* For these events, if you're not calling `app.close()` explicitly, you must opt-in to make them work with system signals such as `SIGTERM`. See Application shutdown below.

> warning **Warning** The lifecycle hooks listed above are not triggered for **request-scoped** classes. Request-scoped classes are not tied to the application lifecycle and their lifespan is unpredictable. They are exclusively created for each request and automatically garbage-collected after the response is sent.

> info **Hint** Execution order of `onModuleInit()` and `onApplicationBootstrap()` directly depends on the order of module imports, awaiting the previous hook.

Usage

Each lifecycle hook is represented by an interface. Interfaces are technically optional because they do not exist after TypeScript compilation. Nonetheless, it's good practice to use them in order to benefit from strong typing and editor tooling. To register a lifecycle hook, implement the appropriate interface. For example, to register a method to be called during module initialization on a particular class (e.g., Controller, Provider or Module), implement the `OnModuleInit` interface by supplying an `onModuleInit()` method, as shown below:

```
@@filename()
import { Injectable, OnModuleInit } from '@nestjs/common';

@Injectable()
export class UsersService implements OnModuleInit {
  onModuleInit() {
    console.log(`The module has been initialized.`);
  }
}
@@switch
import { Injectable } from '@nestjs/common';

@Injectable()
export class UsersService {
  onModuleInit() {
    console.log(`The module has been initialized.`);
  }
}
```

Asynchronous initialization

Both the `OnModuleInit` and `OnApplicationBootstrap` hooks allow you to defer the application initialization process (return a `Promise` or mark the method as `async` and `await` an asynchronous method completion in the method body).

```
@@filename()
async onModuleInit(): Promise<void> {
  await this.fetch();
}
@@switch
async onModuleInit() {
```

```
    await this.fetch();
  }
```

Application shutdown

The `onModuleDestroy()`, `beforeApplicationShutdown()` and `onApplicationShutdown()` hooks are called in the terminating phase (in response to an explicit call to `app.close()` or upon receipt of system signals such as SIGTERM if opted-in). This feature is often used with Kubernetes to manage containers' lifecycles, by Heroku for dynos or similar services.

Shutdown hook listeners consume system resources, so they are disabled by default. To use shutdown hooks, you **must enable listeners** by calling `enableShutdownHooks()`:

```
import { NestFactory } from '@nestjs/core';
import { AppModule } from './app.module';

async function bootstrap() {
  const app = await NestFactory.create(AppModule);

  // Starts listening for shutdown hooks
  app.enableShutdownHooks();

  await app.listen(3000);
}
bootstrap();
```

> warning **warning** Due to inherent platform limitations, NestJS has limited support for application shutdown hooks on Windows. You can expect `SIGINT` to work, as well as `SIGBREAK` and to some extent `SIGHUP` - read more. However `SIGTERM` will never work on Windows because killing a process in the task manager is unconditional, "i.e., there's no way for an application to detect or prevent it". Here's some relevant documentation from libuv to learn more about how `SIGINT`, `SIGBREAK` and others are handled on Windows. Also, see Node.js documentation of Process Signal Events

> info **Info** `enableShutdownHooks` consumes memory by starting listeners. In cases where you are running multiple Nest apps in a single Node process (e.g., when running parallel tests with Jest), Node may complain about excessive listener processes. For this reason, `enableShutdownHooks` is not enabled by default. Be aware of this condition when you are running multiple instances in a single Node process.

When the application receives a termination signal it will call any registered `onModuleDestroy()`, `beforeApplicationShutdown()`, then `onApplicationShutdown()` methods (in the sequence described above) with the corresponding signal as the first parameter. If a registered function awaits an asynchronous call (returns a promise), Nest will not continue in the sequence until the promise is resolved or rejected.

```
@@filename()
@Injectable()
```

```
class UsersService implements OnApplicationShutdown {
  onApplicationShutdown(signal: string) {
    console.log(signal); // e.g. "SIGINT"
  }
}
@@switch
@Injectable()
class UsersService implements OnApplicationShutdown {
  onApplicationShutdown(signal) {
    console.log(signal); // e.g. "SIGINT"
  }
}
```

> info **Info** Calling `app.close()` doesn't terminate the Node process but only triggers the `onModuleDestroy()` and `onApplicationShutdown()` hooks, so if there are some intervals, long-running background tasks, etc. the process won't be automatically terminated.