Logger

Nest comes with a built-in text-based logger which is used during application bootstrapping and several other circumstances such as displaying caught exceptions (i.e., system logging). This functionality is provided via the `Logger` class in the `@nestjs/common` package. You can fully control the behavior of the logging system, including any of the following:

- disable logging entirely
- specify the log level of detail (e.g., display errors, warnings, debug information, etc.)
- override timestamp in the default logger (e.g., use ISO8601 standard as date format)
- completely override the default logger
- customize the default logger by extending it
- make use of dependency injection to simplify composing and testing your application

You can also make use of the built-in logger, or create your own custom implementation, to log your own application-level events and messages.

For more advanced logging functionality, you can make use of any Node.js logging package, such as Winston, to implement a completely custom, production grade logging system.

Basic customization

To disable logging, set the `logger` property to `false` in the (optional) Nest application options object passed as the second argument to the `NestFactory.create()` method.

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

To enable specific logging levels, set the `logger` property to an array of strings specifying the log levels to display, as follows:

```
const app = await NestFactory.create(AppModule, {
  logger: ['error', 'warn'],
});
await app.listen(3000);
```

Values in the array can be any combination of `'log'`, `'error'`, `'warn'`, `'debug'`, and `'verbose'`.

> info **Hint** To disable color in the default logger's messages, set the `NO_COLOR` environment variable to some non-empty string.

Custom implementation

You can provide a custom logger implementation to be used by Nest for system logging by setting the value of the `logger` property to an object that fulfills the `LoggerService` interface. For example, you can tell Nest to use the built-in global JavaScript `console` object (which implements the `LoggerService` interface), as follows:

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

Implementing your own custom logger is straightforward. Simply implement each of the methods of the `LoggerService` interface as shown below.

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

export class MyLogger implements LoggerService {
  /**
   * Write a 'log' level log.
   */
  log(message: any, ...optionalParams: any[]) {}

  /**
   * Write an 'error' level log.
   */
  error(message: any, ...optionalParams: any[]) {}

  /**
   * Write a 'warn' level log.
   */
  warn(message: any, ...optionalParams: any[]) {}

  /**
   * Write a 'debug' level log.
   */
  debug?(message: any, ...optionalParams: any[]) {}

  /**
   * Write a 'verbose' level log.
   */
  verbose?(message: any, ...optionalParams: any[]) {}
}
```

You can then supply an instance of `MyLogger` via the `logger` property of the Nest application options object.

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

This technique, while simple, doesn't utilize dependency injection for the `MyLogger` class. This can pose some challenges, particularly for testing, and limit the reusability of `MyLogger`. For a better solution, see the Dependency Injection section below.

Extend built-in logger

Rather than writing a logger from scratch, you may be able to meet your needs by extending the built-in `ConsoleLogger` class and overriding selected behavior of the default implementation.

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

export class MyLogger extends ConsoleLogger {
  error(message: any, stack?: string, context?: string) {
    // add your tailored logic here
    super.error(...arguments);
  }
}
```

You can use such an extended logger in your feature modules as described in the Using the logger for application logging section below.

You can tell Nest to use your extended logger for system logging by passing an instance of it via the `logger` property of the application options object (as shown in the Custom implementation section above), or by using the technique shown in the Dependency Injection section below. If you do so, you should take care to call `super`, as shown in the sample code above, to delegate the specific log method call to the parent (built-in) class so that Nest can rely on the built-in features it expects.

Dependency injection

For more advanced logging functionality, you'll want to take advantage of dependency injection. For example, you may want to inject a `ConfigService` into your logger to customize it, and in turn inject your custom logger into other controllers and/or providers. To enable dependency injection for your custom logger, create a class that implements `LoggerService` and register that class as a provider in some module. For example, you can

1. Define a `MyLogger` class that either extends the built-in `ConsoleLogger` or completely overrides it, as shown in previous sections. Be sure to implement the `LoggerService` interface.
2. Create a `LoggerModule` as shown below, and provide `MyLogger` from that module.

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

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

With this construct, you are now providing your custom logger for use by any other module. Because your `MyLogger` class is part of a module, it can use dependency injection (for example, to inject a `ConfigService`). There's one more technique needed to provide this custom logger for use by Nest for system logging (e.g., for bootstrapping and error handling).

Because application instantiation (`NestFactory.create()`) happens outside the context of any module, it doesn't participate in the normal Dependency Injection phase of initialization. So we must ensure that at least one application module imports the `LoggerModule` to trigger Nest to instantiate a singleton instance of our `MyLogger` class.

We can then instruct Nest to use the same singleton instance of `MyLogger` with the following construction:

```
const app = await NestFactory.create(AppModule, {
  bufferLogs: true,
});
app.useLogger(app.get(MyLogger));
await app.listen(3000);
```

> info **Note** In the example above, we set the `bufferLogs` to `true` to make sure all logs will be buffered until a custom logger is attached (`MyLogger` in this case) and the application initialisation process either completes or fails. If the initialisation process fails, Nest will fallback to the original `ConsoleLogger` to print out any reported error messages. Also, you can set the `autoFlushLogs` to `false` (default `true`) to manually flush logs (using the `Logger#flush()` method).

Here we use the `get()` method on the `NestApplication` instance to retrieve the singleton instance of the `MyLogger` object. This technique is essentially a way to "inject" an instance of a logger for use by Nest. The `app.get()` call retrieves the singleton instance of `MyLogger`, and depends on that instance being first injected in another module, as described above.

You can also inject this `MyLogger` provider in your feature classes, thus ensuring consistent logging behavior across both Nest system logging and application logging. See Using the logger for application logging and Injecting a custom logger below for more information.

Using the logger for application logging

We can combine several of the techniques above to provide consistent behavior and formatting across both Nest system logging and our own application event/message logging.

A good practice is to instantiate `Logger` class from `@nestjs/common` in each of our services. We can supply our service name as the `context` argument in the `Logger` constructor, like so:

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

```
@Injectable()
class MyService {
  private readonly logger = new Logger(MyService.name);

  doSomething() {
    this.logger.log('Doing something...');
  }
}
```

In the default logger implementation, `context` is printed in the square brackets, like `NestFactory` in the example below:

```
[Nest] 19096   - 12/08/2019, 7:12:59 AM   [NestFactory] Starting Nest
application...
```

If we supply a custom logger via `app.useLogger()`, it will actually be used by Nest internally. That means that our code remains implementation agnostic, while we can easily substitute the default logger for our custom one by calling `app.useLogger()`.

That way if we follow the steps from the previous section and call `app.useLogger(app.get(MyLogger))`, the following calls to `this.logger.log()` from `MyService` would result in calls to method `log` from `MyLogger` instance.

This should be suitable for most cases. But if you need more customization (like adding and calling custom methods), move to the next section.

Injecting a custom logger

To start, extend the built-in logger with code like the following. We supply the `scope` option as configuration metadata for the `ConsoleLogger` class, specifying a transient scope, to ensure that we'll have a unique instance of the `MyLogger` in each feature module. In this example, we do not extend the individual `ConsoleLogger` methods (like `log()`, `warn()`, etc.), though you may choose to do so.

```
import { Injectable, Scope, ConsoleLogger } from '@nestjs/common';

@Injectable({ scope: Scope.TRANSIENT })
export class MyLogger extends ConsoleLogger {
  customLog() {
    this.log('Please feed the cat!');
  }
}
```

Next, create a `LoggerModule` with a construction like this:

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

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

Next, import the `LoggerModule` into your feature module. Since we extended default `Logger` we have the convenience of using `setContext` method. So we can start using the context-aware custom logger, like this:

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

@Injectable()
export class CatsService {
  private readonly cats: Cat[] = [];

  constructor(private myLogger: MyLogger) {
    // Due to transient scope, CatsService has its own unique instance of
MyLogger,
    // so setting context here will not affect other instances in other
services
    this.myLogger.setContext('CatsService');
  }

  findAll(): Cat[] {
    // You can call all the default methods
    this.myLogger.warn('About to return cats!');
    // And your custom methods
    this.myLogger.customLog();
    return this.cats;
  }
}
```

Finally, instruct Nest to use an instance of the custom logger in your `main.ts` file as shown below. Of course in this example, we haven't actually customized the logger behavior (by extending the `Logger` methods like `log()`, `warn()`, etc.), so this step isn't actually needed. But it **would** be needed if you added custom logic to those methods and wanted Nest to use the same implementation.

```
const app = await NestFactory.create(AppModule, {
  bufferLogs: true,
});
app.useLogger(new MyLogger());
await app.listen(3000);
```

> info **Hint** Alternatively, instead of setting `bufferLogs` to `true`, you could temporarily disable the logger with `logger: false` instruction. Be mindful that if you supply `logger: false` to `NestFactory.create`, nothing will be logged until you call `useLogger`, so you may miss some important initialization errors. If you don't mind that some of your initial messages will be logged with the default logger, you can just omit the `logger: false` option.

Use external logger

Production applications often have specific logging requirements, including advanced filtering, formatting and centralized logging. Nest's built-in logger is used for monitoring Nest system behavior, and can also be useful for basic formatted text logging in your feature modules while in development, but production applications often take advantage of dedicated logging modules like Winston. As with any standard Node.js application, you can take full advantage of such modules in Nest.