Logger

발표 날짜	@2021년 9월 11일
발표 여부	☑
발표자	최상희

Logger

- Nest는 내장 텍스트 기반 로거를 제공
- `@nest/common` 패키지의 `Logger` 클래스를 통해 제공
- 아래 항목을 통해 로깅 시스템의 동작을 완벽 제어 가능
  - 로깅을 완전히 비활성화
  - 세부 로그 수준 지정 (예: 디스플레이 오류, 경고, 디버그 정보 등)
  - 기본 로거의 타임스탬프 재정의
  - 기본 로거를 완전히 무시
  - 기본 로거를 확장하여 사용자 지정
  - 종속성 주입을 사용하여 애플리케이션 작성 및 테스트를 단순화
- 고급 로깅 기능을 위해 `Winston` 과 같은 로깅 패키지를 사용하여 완전한 맞춤형 로깅 시스템 구현 가능

Basic customization

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

- `NestFactory.create()` 메소드에 두번째 인수로 전달되는 옵션 객체에서 `logger: false` 를 통해 로깅 사용 중지 가능

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

- 특정 로깅 수준을 사용하려면 로그 수준을 지정하는 문자열 배열로 `logger` 속성을 설정
  - 배열의 값은 `log`, `error`, `warn`, `debug`, `verbose` 의 조합으로 가능

Custom implementation

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

- `logger` 속성의 값을 `LoggerService` 인터페이스를 충족하는 객체로 설정하여 사용자 정의 로거 구현이 가능
- 위 예제는 내장된 전역 자바 스크립트 `console` 객체(`LoggerService` 인터페이스를 구현함)를 사용하도록 설정

```
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[]) {}
}
```

- 사용자 정의 로거를 구현하는 것은 위와 같이 `LoggerService` 의 각 메소드를 구현하기만 하면 끝

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

- 이후에는 위와 같이 해당 인스턴스를 제공하면 적용 완료
- 이 기술은 간단하지만 종속성 주입을 사용하지 않았음. 이로 인해 테스트시 몇가지 문제가 발생할 수 있으며 `MyLogger` 의 재사용 가능성이 제한됨. 저어어기 아래에서 종속성 주입에 관해 알려주겠음.

Extend built-in logger

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

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

- 처음부터 로거를 작성하는 대신에 `ConsoleLogger` 클래스를 확장하고 기본 구현된 메소드를 재정의하여 커스텀 가능

Dependency Injection

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

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

- 위와 같이 `LoggerModule` 을 생성하여 해당 모듈에서 `MyLogger` 를 제공하도록 설정
- 이제는 MyLogger가 모듈의 일부가 되었기 때문에 종속성 주입이 가능해짐
- 그러나 `NestFactory.create()` 는 모듈 컨텍스트 외부에서 발생하기 때문에 일반적인 종속성 주입 단계에 참여하지 않음. 따라서 적어도 하나의 애플리케이션 모듈이 `LoggerModule` 을 가져와 Nest를 트리거하여 `MyLogger` 클래스의 단일 인스턴스를 인스턴스화하도록 해야함

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

- 위 방법은 `NestApplication` 인스턴스의 `get()` 메소드를 사용하여 `MyLogger` 객체의 싱글톤 인스턴스를 검색함
- 로거 인스턴스를 주입하는 방법임.
- 추가 내용
  - `bufferLogs: true` 를 설정함으로써 애플리케이션 초기화 프로세스의 로그가 버퍼링 됨
  - 초기화 프로세스가 실패하면 Nest는 디폴트 로거인 `ConsoleLogger` 를 사용

Using the logger for application logging

- 위의 기술을 결합하여 Nest 시스템 로깅과 자체 애플리케이션 로깅 모두에서 일관된 동작과 형식을 제공 가능

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

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

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

- 각 서비스의 `@nestjs/common` 에서 `Logger` 클래스를 인스턴스화하는 것이 좋음
- 위와 같이 `Logger` 생성자에서 `context` 인수로 서비스 이름을 제공 가능

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

- `context` 는 위의 `NestFactory` 와 같이 대괄호 안에 인쇄
- `app.userLogger()` 를 통해 커스텀 로거를 제공하면 실제로 Nest에서 내부적으로 사용. 즉, 이렇게 해두면 `MyService` 에서 `this.logger.log` 를 호출하면 `MyLogger` 인스턴스에서 메소드 `log` 가 호출됨
- 대부분의 경우 이렇게 처리하면 되지만 사용자 지정 메소드 추가 및 호출과 같은 추가 커스터마이징이 필요한 경우 다음 섹션으로 이동

Injecting a custom logger

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

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

- 내장 로거를 확장하고 각 기능 모듈에서 `MyLogger` 의 고유한 인스턴스를 갖도록 `scope: Scope.TRANSIENT` 로 지정 ⇒ 싱글 인스턴스로 하면 안되나? ⇒ 아 이러면 context 적용을 못 시키는구나

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

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

- `MyLogger` 를 모듈화함

```
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;
  }
}
```

- 요렇게 `customLog()` 메소드 호출 성공!
- 기본 로거를 확장하였으므로 당연히 `warn` 메소드 또한 사용 가능

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

- `app.useLogger` 를 통해 커스텀 로거의 인스턴스를 사용하도록 지시
- `bufferLogs` 를 `true` 로 설정하는 대신 `logger: false` 명령으로 로거를 일시적으로 비활성화 할 수 있음
- 이럴 경우 초기화 과정이 기록되지 않으므로 중요한 초기화 오류를 놓칠 수 있음
- 초기 메시지중 일부가 기본 로거로 기록된다는 점이 마음에 들지 않으면 위 스킬로 패스 가능

Use external logger

- Winston을 씁시다