File upload

In order to handle file uploading, Nest makes use of **multer** middleware. This middleware is fully configurable and you can adjust its behavior to your application requirements.

Multer is middleware for handling `multipart/form-data`, which is primarily used for uploading files.

> **WARNING**
> Multer will not process any form which is not multipart (`multipart/form-data`). Besides, this package won't work with the `FastifyAdapter` .

Basic example

When we want to upload a single file, we simply tie `FileInterceptor()` to the handler, and then, pull outs `file` from the `request` using `@UploadedFile()` decorator.

JS

```
@Post('upload')
@UseInterceptors(FileInterceptor('file'))
uploadFile(@UploadedFile() file) {
  console.log(file);
}
```

> **HINT**
> `FileInterceptor()` decorator is exported from `@nestjs/platform-express` package while `@UploadedFile()` from `@nestjs/common` .

The `FileInterceptor()` takes two arguments, a `fieldName` (points to field from HTML form that holds a file) and optional `options` object. These `MulterOptions` are equivalent to those passed into multer constructor (more details **here**)

Array of files

In order to upload an array of files, we use `FilesInterceptor()` . This interceptor takes three arguments. A `fieldName` (that remains the same), `maxCount` which is a maximum number of files that can be uploaded at the same time, and optional `MulterOptions` object. Additionally, to pick files from `request` object, we use `@UploadedFiles()` decorator

JS

```
@Post('upload')
@UseInterceptors(FilesInterceptor('files'))
uploadFile(@UploadedFiles() files) {
  console.log(files);
}
```

> **HINT**
> `FilesInterceptor()` decorator is exported from `@nestjs/platform-express` package while `@UploadedFiles()` from `@nestjs/common`.

Multiple files

To upload multiple fields (all with different keys), we use `FileFieldsInterceptor()` decorator.

JS

```
@Post('upload')
@UseInterceptors(FileFieldsInterceptor([
  { name: 'avatar', maxCount: 1 },
  { name: 'background', maxCount: 1 },
]))
uploadFile(@UploadedFiles() files) {
  console.log(files);
}
```

Any files

To upload any fields (all with different keys, but you don't have to know them), we use `AnyFilesInterceptor()` decorator.

JS

```
@Post('upload')
@UseInterceptors(AnyFilesInterceptor())
uploadFile(@UploadedFiles() files) {
  console.log(files);
}
```

Default options

To customize **multer** behavior, you can register the `MulterModule`. We support all options listed **here**.

```
MulterModule.register({
  dest: '/upload',
});
```

Async configuration

Quite often you might want to asynchronously pass your module options instead of passing them beforehand. In such case, use `registerAsync()` method, that provides a couple of various ways to deal with async data.

First possible approach is to use a factory function:

```
MulterModule.registerAsync({
  useFactory: () => ({
    dest: '/upload',
  }),
});
```

Obviously, our factory behaves like every other one (might be `async` and is able to inject dependencies through `inject`).

```
MulterModule.registerAsync({
  imports: [ConfigModule],
  useFactory: async (configService: ConfigService) => ({
    dest: configService.getString('MULTER_DEST'),
  }),
  inject: [ConfigService],
});
```

Alternatively, you are able to use class instead of a factory.

```
MulterModule.registerAsync({
  useClass: MulterConfigService,
});
```

Above construction will instantiate `MulterConfigService` inside `MulterModule` and will leverage it to create options object. The `MulterConfigService` has to implement `MulterOptionsFactory` interface.

```
@Injectable()
class MulterConfigService implements MulterOptionsFactory {
  createMulterOptions(): MulterModuleOptions {
    return {
      dest: '/upload',
    };
  }
}
```

In order to prevent the creation of `MulterConfigService` inside `MulterModule` and use a provider imported from a different module, you can use the `useExisting` syntax.

```
MulterModule.registerAsync({
  imports: [ConfigModule],
  useExisting: ConfigService,
});
```

It works the same as `useClass` with one critical difference - `MulterModule` will lookup imported modules to reuse already created `ConfigService`, instead of instantiating it on its own.

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