



















[Get 10 free Adobe Stock photos. Start downloading amazing royalty-free stock photos](#)

[today.ADS VIA CARBON](#)

## File upload

To handle file uploading, Nest provides a built-in module based on the **multer** middleware package for Express. Multer handles data posted in the `multipart/form-data` format, which is primarily used for uploading files via an HTTP `POST` request. This module is fully configurable and you can adjust its behavior to your application requirements.

**WARNING** Multer cannot process data which is not in the supported multipart format (`multipart/form-data`). Also, note that this package is not compatible with the `FastifyAdapter`.

For better type safety, let's install Multer typings package:

```
$ npm i -D @types/multer
```

With this package installed, we can now use the `Express.Multer.File` type (you can import this type as follows: `import { Express } from 'express'`).

## Basic example#

To upload a single file, simply tie the `FileInterceptor()` interceptor to the route handler and extract `file` from the `request` using the `@UploadedFile()` decorator.

**JS**

```

@Post('upload')

@UseInterceptors(FileInterceptor('file'))

uploadFile(@UploadedFile() file: Express.Multer.File) {

    console.log(file);

}

```

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

The `FileInterceptor()` decorator takes two arguments:

- `fieldName`: string that supplies the name of the field from the HTML form that holds a file
- `options`: optional object of type `MulterOptions`. This is the same object used by the multer constructor (more details [here](#)).

**WARNING** `FileInterceptor()` may not be compatible with third party cloud providers like Google Firebase or others.

## File validation#

Often times it can be useful to validate incoming file metadata, like file size or file mime-type. For this, you can create your own **Pipe** and bind it to the parameter annotated with the `UploadedFile` decorator. The example below demonstrates how a basic file size validator pipe could be implemented:

```

// ...
import { PipeTransform, Injectable } from '@nestjs/core';
import { Multer } from 'multer';
import { FileInterceptor } from '@nestjs/platform-express';
import { UploadedFile } from '@nestjs/common';

@Injectable()
export class FileSizeValidator implements PipeTransform {
  transform(value: UploadedFile, _metadata: any): UploadedFile {
    if (value.size > 1024 * 1024) {
      throw new Error('File size is too large');
    }
    return value;
  }
}

```



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

@Injectable()

export class FileSizeValidationPipe implements PipeTransform {

  transform(value: any, metadata: ArgumentMetadata) {

    // "value" is an object containing the file's attributes and metadata

    const oneKb = 1000;

    return value.size < oneKb;

  }

}
```

Nest provides a built-in pipe to handle common use cases and facilitate/standardize the addition of new ones. This pipe is called [ParseFilePipe](#), and you can use it as follows:

```
@Post('file')

uploadFileAndPassValidation(
```

```
@Body() body: SampleDto,

@UploadedFile(

  new ParseFilePipe({

    validators: [

      // ... Set of file validator instances here

    ]

  })

)

file: Express.Multer.File,
) {

  return {

    body,

    file: file.buffer.toString(),

  };
};
```

```
}
```

As you can see, it's required to specify an array of file validators that will be executed by the `ParseFilePipe`. We'll discuss the interface of a validator, but it's worth mentioning this pipe also has two additional **optional** options:

`errorHttpStatusCode`

The HTTP status code to be thrown in case **any** validator fails. Default is `400` (BAD REQUEST)

`exceptionFactory`

A factory which receives the error message and returns an error.

Now, back to the `FileValidator` interface. To integrate validators with this pipe, you have to either use built-in implementations or provide your own custom `FileValidator`. See example below:

```
export abstract class FileValidator<TValidationOptions = Record<string, any>>
{
    constructor(protected readonly validationOptions: TValidationOptions) {}
}
```

```

/**
 * Indicates if this file should be considered valid, according to the
options passed in the constructor.

 * @param file the file from the request object

 */

abstract isValid(file?: any): boolean | Promise<boolean>;

/**
 * Builds an error message in case the validation fails.

 * @param file the file from the request object

 */

abstract buildErrorMessage(file: any): string;
}

```

**HINT** The `FileValidator` interfaces supports async validation via its `isValid` function. To leverage type security, you can also type the `file` parameter as `Express.Multer.File` in case you are using express (default) as a driver.



`FileValidator` is a regular class that has access to the file object and validates it according to the options provided by the client. Nest has two built-in `FileValidator` implementations you can use in your project:

- `MaxFileSizeValidator` - Checks if a given file's size is less than the provided value (measured in `bytes`)
- `FileTypeValidator` - Checks if a given file's mime-type matches the given value.

**WARNING** To verify file type, `FileTypeValidator` class uses the type as detected by `multer`. By default, `multer` derives file type from file extension on user's device. However, it does not check actual file contents. As files can be renamed to arbitrary extensions, consider using a custom implementation (like checking the file's `magic number`) if your app requires a safer solution.

To understand how these can be used in conjunction with the aforementioned `FileParsePipe`, we'll use an altered snippet of the last presented example:

```
@UploadedFile(  
  
  new ParseFilePipe({  
  
    validators: [  
  
      new MaxFileSizeValidator({ maxSize: 1000 }),  
  
      new FileTypeValidator({ fileType: 'image/jpeg' }),  
  
    ],  
  
  })),
```

```
)
```

```
file: Express.Multer.File,
```

**HINT** If the number of validators increase largely or their options are cluttering the file, you can define this array in a separate file and import it here as a named constant like `fileValidators`.

Finally, you can use the special `ParseFilePipeBuilder` class that lets you compose & construct your validators. By using it as shown below you can avoid manual instantiation of each validator and just pass their options directly:

```
@UploadedFile(  
  
  new ParseFilePipeBuilder()  
  
    .addFileTypeValidator({  
  
      fileType: 'jpeg',  
  
    })  
  
    .addMaxSizeValidator({  
  
      maxSize: 1000  
  
    })  
  
  })
```

```

        .build({

            errorHttpStatusCode: HttpStatus.UNPROCESSABLE_ENTITY

        }),

    )

file: Express.Multer.File,

```

## Array of files#

To upload an array of files (identified with a single field name), use the `FilesInterceptor()` decorator (note the plural **Files** in the decorator name). This decorator takes three arguments:

- `fieldName`: as described above
- `maxCount`: optional number defining the maximum number of files to accept
- `options`: optional `MulterOptions` object, as described above

When using `FilesInterceptor()`, extract files from the `request` with the `@UploadedFiles()` decorator.

JS

```

@Post('upload')

@UseInterceptors(FilesInterceptor('files'))

```

```
uploadFile(@UploadedFiles() files: Array<Express.Multer.File>) {

    console.log(files);

}
```

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

## Multiple files#

To upload multiple files (all with different field name keys), use the `FileFieldsInterceptor()` decorator. This decorator takes two arguments:

- `uploadedFields`: an array of objects, where each object specifies a required `name` property with a string value specifying a field name, as described above, and an optional `maxCount` property, as described above
- `options`: optional `MulterOptions` object, as described above

When using `FileFieldsInterceptor()`, extract files from the `request` with the `@UploadedFiles()` decorator.

**JS**

```
@Post('upload')

@UseInterceptors(FileFieldsInterceptor([

    { name: 'avatar', maxCount: 1 },
```



```

    { name: 'background', maxCount: 1 },

  ]))

  uploadFile(@UploadedFiles() files: { avatar?: Express.Multer.File[],
background?: Express.Multer.File[] }) {

    console.log(files);

  }

```

## Any files#

To upload all fields with arbitrary field name keys, use the `AnyFilesInterceptor()` decorator. This decorator can accept an optional `options` object as described above.

When using `AnyFilesInterceptor()`, extract files from the `request` with the `@UploadedFiles()` decorator.

JS

```

@Post('upload')

@UseInterceptors(AnyFilesInterceptor())

uploadFile(@UploadedFiles() files: Array<Express.Multer.File>) {

  console.log(files);
}

```

```
}
```

## Default options#

You can specify multer options in the file interceptors as described above. To set default options, you can call the static `register()` method when you import the `MulterModule`, passing in supported options. You can use all options listed [here](#).

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

**HINT** The `MulterModule` class is exported from the `@nestjs/platform-express` package.

## Async configuration#

When you need to set `MulterModule` options asynchronously instead of statically, use the `registerAsync()` method. As with most dynamic modules, Nest provides several techniques to deal with async configuration.

One technique is to use a factory function:

```
MulterModule.registerAsync({
```

```

    useFactory: () => ({

        dest: './upload',

    }),

});

```

Like other **factory providers**, our factory function can be `async` and can inject dependencies through `inject`.

```

MulterModule.registerAsync({

    imports: [ConfigModule],

    useFactory: async (configService: ConfigService) => ({

        dest: configService.get<string>('MULTER_DEST'),

    }),

    inject: [ConfigService],

});

```

Alternatively, you can configure the `MulterModule` using a class instead of a factory, as shown below:

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

The construction above instantiates `MulterConfigService` inside `MulterModule`, using it to create the required options object. Note that in this example, the `MulterConfigService` has to implement the `MulterOptionsFactory` interface, as shown below. The `MulterModule` will call the `createMulterOptions()` method on the instantiated object of the supplied class.

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



```
}
```

If you want to reuse an existing options provider instead of creating a private copy inside the `MulterModule`, use the `useExisting` syntax.

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

## Example#

A working example is available [here](#).

## 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 Sponsors



# SANOFI



**Sponsors / Partners**

[Become a sponsor](#)

**Join our Newslett**