Versioning

> info **Hint** This chapter is only relevant to HTTP-based applications.

Versioning allows you to have **different versions** of your controllers or individual routes running within the same application. Applications change very often and it is not unusual that there are breaking changes that you need to make while still needing to support the previous version of the application.

There are 4 types of versioning that are supported:

`URI Versioning`	The version will be passed within the URI of the request (default)
`Header Versioning`	A custom request header will specify the version
`Media Type Versioning`	The `Accept` header of the request will specify the version
`Custom Versioning`	Any aspect of the request may be used to specify the version(s). A custom function is provided to extract said version(s).

URI Versioning Type

URI Versioning uses the version passed within the URI of the request, such as `https://example.com/v1/route` and `https://example.com/v2/route`.

> warning **Notice** With URI Versioning the version will be automatically added to the URI after the global path prefix (if one exists), and before any controller or route paths.

To enable URI Versioning for your application, do the following:

```
@@filename(main)
const app = await NestFactory.create(AppModule);
// or "app.enableVersioning()"
app.enableVersioning({
  type: VersioningType.URI,
});
await app.listen(3000);
```

> warning **Notice** The version in the URI will be automatically prefixed with `v` by default, however the prefix value can be configured by setting the `prefix` key to your desired prefix or `false` if you wish to disable it.

> info **Hint** The `VersioningType` enum is available to use for the `type` property and is imported from the `@nestjs/common` package.

Header Versioning Type

Header Versioning uses a custom, user specified, request header to specify the version where the value of the header will be the version to use for the request.

Example HTTP Requests for Header Versioning:

To enable **Header Versioning** for your application, do the following:

```
@@filename(main)
const app = await NestFactory.create(AppModule);
app.enableVersioning({
  type: VersioningType.HEADER,
  header: 'Custom-Header',
});
await app.listen(3000);
```

The `header` property should be the name of the header that will contain the version of the request.

> info **Hint** The `VersioningType` enum is available to use for the `type` property and is imported from the `@nestjs/common` package.

Media Type Versioning Type

Media Type Versioning uses the `Accept` header of the request to specify the version.

Within the `Accept` header, the version will be separated from the media type with a semi-colon, `;`. It should then contain a key-value pair that represents the version to use for the request, such as `Accept: application/json;v=2`. They key is treated more as a prefix when determining the version will to be configured to include the key and separator.

To enable **Media Type Versioning** for your application, do the following:

```
@@filename(main)
const app = await NestFactory.create(AppModule);
app.enableVersioning({
  type: VersioningType.MEDIA_TYPE,
  key: 'v=',
});
await app.listen(3000);
```

The `key` property should be the key and separator of the key-value pair that contains the version. For the example `Accept: application/json;v=2`, the `key` property would be set to `v=`.

> info **Hint** The `VersioningType` enum is available to use for the `type` property and is imported from the `@nestjs/common` package.

Custom Versioning Type

Custom Versioning uses any aspect of the request to specify the version (or versions). The incoming request is analyzed using an `extractor` function that returns a string or array of strings.

If multiple versions are provided by the requester, the extractor function can return an array of strings, sorted in order of greatest/highest version to smallest/lowest version. Versions are matched to routes in order from highest to lowest.

If an empty string or array is returned from the `extractor`, no routes are matched and a 404 is returned.

For example, if an incoming request specifies it supports versions `1`, `2`, and `3`, the `extractor` **MUST** return `[3, 2, 1]`. This ensures that the highest possible route version is selected first.

If versions `[3, 2, 1]` are extracted, but routes only exist for version `2` and `1`, the route that matches version `2` is selected (version `3` is automatically ignored).

> warning **Notice** Selecting the highest matching version based on the array returned from `extractor` > **does not reliably work** with the Express adapter due to design limitations. A single version (either a string or array of 1 element) works just fine in Express. Fastify correctly supports both highest matching version selection and single version selection.

To enable **Custom Versioning** for your application, create an `extractor` function and pass it into your application like so:

```
@@filename(main)
// Example extractor that pulls out a list of versions from a custom
header and turns it into a sorted array.
// This example uses Fastify, but Express requests can be processed in a
similar way.
const extractor = (request: FastifyRequest): string | string[] =>
  [request.headers['custom-versioning-field'] ?? '']
     .flatMap(v => v.split(','))
     .filter(v => !!v)
     .sort()
     .reverse()

const app = await NestFactory.create(AppModule);
app.enableVersioning({
  type: VersioningType.CUSTOM,
  extractor,
});
await app.listen(3000);
```

Usage

Versioning allows you to version controllers, individual routes, and also provides a way for certain resources to opt-out of versioning. The usage of versioning is the same regardless of the Versioning Type your application uses.

> warning **Notice** If versioning is enabled for the application but the controller or route does not specify the version, any requests to that controller/route will be returned a `404` response status.

> Similarly, if a request is received containing a version that does not have a corresponding controller or route, it will also be returned a `404` response status.

Controller versions

A version can be applied to a controller, setting the version for all routes within the controller.

To add a version to a controller do the following:

```
@@filename(cats.controller)
@Controller({
  version: '1',
})
export class CatsControllerV1 {
  @Get('cats')
  findAll(): string {
    return 'This action returns all cats for version 1';
  }
}
@@switch
@Controller({
  version: '1',
})
export class CatsControllerV1 {
  @Get('cats')
  findAll() {
    return 'This action returns all cats for version 1';
  }
}
```

Route versions

A version can be applied to an individual route. This version will override any other version that would effect the route, such as the Controller Version.

To add a version to an individual route do the following:

```
@@filename(cats.controller)
import { Controller, Get, Version } from '@nestjs/common';

@Controller()
export class CatsController {
  @Version('1')
  @Get('cats')
  findAllV1(): string {
    return 'This action returns all cats for version 1';
  }

  @Version('2')
  @Get('cats')
```

```
  findAllV2(): string {
    return 'This action returns all cats for version 2';
  }
}
@@switch
import { Controller, Get, Version } from '@nestjs/common';

@Controller()
export class CatsController {
  @Version('1')
  @Get('cats')
  findAllV1() {
    return 'This action returns all cats for version 1';
  }

  @Version('2')
  @Get('cats')
  findAllV2() {
    return 'This action returns all cats for version 2';
  }
}
```

Multiple versions

Multiple versions can be applied to a controller or route. To use multiple versions, you would set the version to be an Array.

To add multiple versions do the following:

```
@@filename(cats.controller)
@Controller({
  version: ['1', '2'],
})
export class CatsController {
  @Get('cats')
  findAll(): string {
    return 'This action returns all cats for version 1 or 2';
  }
}
@@switch
@Controller({
  version: ['1', '2'],
})
export class CatsController {
  @Get('cats')
  findAll() {
    return 'This action returns all cats for version 1 or 2';
  }
}
```

Version "Neutral"

Some controllers or routes may not care about the version and would have the same functionality regardless of the version. To accommodate this, the version can be set to `VERSION_NEUTRAL` symbol.

An incoming request will be mapped to a `VERSION_NEUTRAL` controller or route regardless of the version sent in the request in addition to if the request does not contain a version at all.

> warning **Notice** For URI Versioning, a `VERSION_NEUTRAL` resource would not have the version present in the URI.

To add a version neutral controller or route do the following:

```
@@filename(cats.controller)
import { Controller, Get, VERSION_NEUTRAL } from '@nestjs/common';

@Controller({
  version: VERSION_NEUTRAL,
})
export class CatsController {
  @Get('cats')
  findAll(): string {
    return 'This action returns all cats regardless of version';
  }
}
@@switch
import { Controller, Get, VERSION_NEUTRAL } from '@nestjs/common';

@Controller({
  version: VERSION_NEUTRAL,
})
export class CatsController {
  @Get('cats')
  findAll() {
    return 'This action returns all cats regardless of version';
  }
}
```

Global default version

If you do not want to provide a version for each controller/or individual routes, or if you want to have a specific version set as the default version for every controller/route that don't have the version specified, you could set the `defaultVersion` as follows:

```
@@filename(main)
app.enableVersioning({
  // ...
  defaultVersion: '1'
  // or
```

```
  defaultVersion: ['1', '2']
  // or
  defaultVersion: VERSION_NEUTRAL
});
```

Middleware versioning

Middlewares can also use versioning metadata to configure the middleware for a specific route's version. To do so, provide the version number as one of the parameters for the `MiddlewareConsumer.forRoutes()` method:

```
@@filename(app.module)
import { Module, NestModule, MiddlewareConsumer } from '@nestjs/common';
import { LoggerMiddleware } from './common/middleware/logger.middleware';
import { CatsModule } from './cats/cats.module';
import { CatsController } from './cats/cats.controller';

@Module({
  imports: [CatsModule],
})
export class AppModule implements NestModule {
  configure(consumer: MiddlewareConsumer) {
    consumer
      .apply(LoggerMiddleware)
      .forRoutes({ path: 'cats', method: RequestMethod.GET, version: '2'
});
  }
}
```

With the code above, the `LoggerMiddleware` will only be applied to the version '2' of `/cats` endpoint.

> info **Notice** Middlewares work with any versioning type described in the this section: `URI`, `Header`, `Media Type` or `Custom`.