Cookies

An **HTTP cookie** is a small piece of data stored by the user's browser. Cookies were designed to be a reliable mechanism for websites to remember stateful information. When the user visits the website again, the cookie is automatically sent with the request.

Use with Express (default)

First install the required package (and its types for TypeScript users):

```
$ npm i cookie-parser
$ npm i -D @types/cookie-parser
```

Once the installation is complete, apply the `cookie-parser` middleware as global middleware (for example, in your `main.ts` file).

```
import * as cookieParser from 'cookie-parser';
// somewhere in your initialization file
app.use(cookieParser());
```

You can pass several options to the `cookieParser` middleware:

- `secret` a string or array used for signing cookies. This is optional and if not specified, will not parse signed cookies. If a string is provided, this is used as the secret. If an array is provided, an attempt will be made to unsign the cookie with each secret in order.
- `options` an object that is passed to `cookie.parse` as the second option. See cookie for more information.

The middleware will parse the `Cookie` header on the request and expose the cookie data as the property `req.cookies` and, if a secret was provided, as the property `req.signedCookies`. These properties are name value pairs of the cookie name to cookie value.

When secret is provided, this module will unsign and validate any signed cookie values and move those name value pairs from `req.cookies` into `req.signedCookies`. A signed cookie is a cookie that has a value prefixed with `s:`. Signed cookies that fail signature validation will have the value `false` instead of the tampered value.

With this in place, you can now read cookies from within the route handlers, as follows:

```
@Get()
findAll(@Req() request: Request) {
  console.log(request.cookies); // or "request.cookies['cookieKey']"
  // or console.log(request.signedCookies);
}
```

> info **Hint** The `@Req()` decorator is imported from the `@nestjs/common`, while `Request` from the `express` package.

To attach a cookie to an outgoing response, use the `Response#cookie()` method:

```
@Get()
findAll(@Res({ passthrough: true }) response: Response) {
  response.cookie('key', 'value')
}
```

> warning **Warning** If you want to leave the response handling logic to the framework, remember to set the `passthrough` option to `true`, as shown above. Read more here.

> info **Hint** The `@Res()` decorator is imported from the `@nestjs/common`, while `Response` from the `express` package.

Use with Fastify

First install the required package:

```
$ npm i @fastify/cookie
```

Once the installation is complete, register the `@fastify/cookie` plugin:

```
import fastifyCookie from '@fastify/cookie';

// somewhere in your initialization file
const app = await NestFactory.create<NestFastifyApplication>(
  AppModule,
  new FastifyAdapter(),
);
await app.register(fastifyCookie, {
  secret: 'my-secret', // for cookies signature
});
```

With this in place, you can now read cookies from within the route handlers, as follows:

```
@Get()
findAll(@Req() request: FastifyRequest) {
  console.log(request.cookies); // or "request.cookies['cookieKey']"
}
```

> info **Hint** The `@Req()` decorator is imported from the `@nestjs/common`, while `FastifyRequest` from the `fastify` package.

To attach a cookie to an outgoing response, use the `FastifyReply#setCookie()` method:

```
@Get()
findAll(@Res({ passthrough: true }) response: FastifyReply) {
  response.setCookie('key', 'value')
}
```

To read more about `FastifyReply#setCookie()` method, check out this page.

> warning **Warning** If you want to leave the response handling logic to the framework, remember to set the `passthrough` option to `true`, as shown above. Read more here.

> info **Hint** The `@Res()` decorator is imported from the `@nestjs/common`, while `FastifyReply` from the `fastify` package.

Creating a custom decorator (cross-platform)

To provide a convenient, declarative way of accessing incoming cookies, we can create a custom decorator.

```
import { createParamDecorator, ExecutionContext } from '@nestjs/common';

export const Cookies = createParamDecorator(
  (data: string, ctx: ExecutionContext) => {
    const request = ctx.switchToHttp().getRequest();
    return data ? request.cookies?.[data] : request.cookies;
  },
);
```

The `@Cookies()` decorator will extract all cookies, or a named cookie from the `req.cookies` object and populate the decorated parameter with that value.

With this in place, we can now use the decorator in a route handler signature, as follows:

```
@Get()
findAll(@Cookies('name') name: string) {}
```