# next/server

The `next/server` module provides several exports for server-only helpers, such as Middleware.

## NextMiddleware

Middleware is created by using a `middleware` function that lives inside a `_middleware` file. The Middleware API is based upon the native `Request`, `FetchEvent`, and `Response` objects.

These native Web API objects are extended to give you more control over how you manipulate and configure a response, based on the incoming requests.

The function signature is defined as follows:

```
type NextMiddlewareResult = NextResponse | Response | null | undefined

type NextMiddleware = (
  request: NextRequest,
  event: NextFetchEvent
) => NextMiddlewareResult | Promise<NextMiddlewareResult>
```

It can be imported from `next/server` with the following:

```
import type { NextMiddleware } from 'next/server'
```

The function can be a default export and as such, does **not** have to be named `middleware`. Though this is a convention. Also note that you only need to make the function `async` if you are running asynchronous code.

## NextRequest

The `NextRequest` object is an extension of the native `Request` interface, with the following added methods and properties:

- `cookies` - Has the cookies from the `Request`
- `nextUrl` - Includes an extended, parsed, URL object that gives you access to Next.js specific properties such as `pathname`, `basePath`, `trailingSlash` and `i18n`
- `ip` - Has the IP address of the `Request`
- `ua` - Has the user agent
- `geo` - (Optional) Has the geo location from the `Request`, provided by your hosting platform

You can use the `NextRequest` object as a direct replacement for the native `Request` interface, giving you more control over how you manipulate the request.

`NextRequest` is fully typed and can be imported from `next/server`.

```
import type { NextRequest } from 'next/server'
```

## NextFetchEvent

The `NextFetchEvent` object extends the native `FetchEvent` object, and includes the `waitUntil()` method.

The `waitUntil()` method can be used to prolong the execution of the function, after the response has been sent. In practice this means that you can send a response, then continue the function execution if you have other background work to make.

The `event` object is fully typed and can be imported from `next/server`.

```
import type { NextFetchEvent } from 'next/server'
```

## NextResponse

The `NextResponse` class extends the native `Response` interface, with the following:

### Public methods

Public methods are available on an instance of the `NextResponse` class. Depending on your use case, you can create an instance and assign to a variable, then access the following public methods:

- `cookies` - An object with the cookies in the `Response`
- `cookie()` - Set a cookie in the `Response`
- `clearCookie()` - Accepts a `cookie` and clears it

```
import { NextResponse } from 'next/server'
import type { NextRequest } from 'next/server'

export function middleware(request: NextRequest) {
  // create an instance of the class to access the public methods. This uses
`next()`,
  // you could use `redirect()` or `rewrite()` as well
  let response = NextResponse.next()
  // get the cookies from the request
  let cookieFromRequest = request.cookies['my-cookie']
  // set the `cookie`
  response.cookie('hello', 'world')
  // set the `cookie` with options
  const cookieWithOptions = response.cookie('hello', 'world', {
    path: '/',
    maxAge: 1000 * 60 * 60 * 24 * 7,
    httpOnly: true,
    sameSite: 'strict',
    domain: 'example.com',
  })
  // clear the `cookie`
  response.clearCookie('hello')

  return response
}
```

### Static methods

The following static methods are available on the `NextResponse` class directly:

- `redirect()` - Returns a `NextResponse` with a redirect set
- `rewrite()` - Returns a `NextResponse` with a rewrite set
- `next()` - Returns a `NextResponse` that will continue the middleware chain
- `json()` - A convenience method to create a response that encodes the provided JSON data

```ts
import { NextResponse } from 'next/server'
import type { NextRequest } from 'next/server'

export function middleware(req: NextRequest) {
  // if the request is coming from New York, redirect to the home page
  if (req.geo.city === 'New York') {
    return NextResponse.redirect('/home')
    // if the request is coming from London, rewrite to a special page
  } else if (req.geo.city === 'London') {
    return NextResponse.rewrite('/not-home')
  }

  return NextResponse.json({ message: 'Hello World!' })
}
```

All methods above return a `NextResponse` object that only takes effect if it's returned in the middleware function.

`NextResponse` is fully typed and can be imported from `next/server`.

```ts
import { NextResponse } from 'next/server'
```

### Setting the cookie before a redirect

In order to set the `cookie` *before* a redirect, you can create an instance of `NextResponse`, then access the `cookie` method on the instance, before returning the response.

Note that there is a Chrome bug which means the entire redirect chain **must** be from the same origin, if they are from different origins, then the `cookie` might be missing until a refresh.

```ts
import { NextResponse } from 'next/server'
import type { NextRequest } from 'next/server'

export function middleware(req: NextRequest) {
  const res = NextResponse.redirect('/') // creates an actual instance
  res.cookie('hello', 'world') // can be called on an instance
  return res
}
```

### Why does redirect use 307 and 308?

When using `redirect()` you may notice that the status codes used are `307` for a temporary redirect, and `308` for a permanent redirect. While traditionally a `302` was used for a temporary redirect, and a `301` for a permanent redirect, many browsers changed the request method of the redirect, from a `POST` to `GET` request when using a `302`, regardless of the origins request method.

Taking the following example of a redirect from `/users` to `/people`, if you make a `POST` request to `/users` to create a new user, and are conforming to a `302` temporary redirect, the request method will be changed from a `POST` to a `GET` request. This doesn't make sense, as to create a new user, you should be making a `POST` request to `/people`, and not a `GET` request.

The introduction of the `307` status code means that the request method is preserved as `POST`.

- `302` - Temporary redirect, will change the request method from `POST` to `GET`
- `307` - Temporary redirect, will preserve the request method as `POST`

The `redirect()` method uses a `307` by default, instead of a `302` temporary redirect, meaning your requests will *always* be preserved as `POST` requests.

### How do I access Environment Variables?

`process.env` can be used to access Environment Variables from Middleware. These are evaluated at build time, so only environment variables *actually* used will be included.

Any variables in `process.env` must be accessed directly, and **cannot** be destructured:

```
// Accessed directly, and not destructured works. process.env.NODE_ENV is
`"development"` or `"production"`
console.log(process.env.NODE_ENV)
// This will not work
const { NODE_ENV } = process.env
// NODE_ENV is `undefined`
console.log(NODE_ENV)
// process.env is `{}`
console.log(process.env)
```

## Related

**Edge Runtime** Learn more about the supported Web APIs available.
**Middleware** Run code before a request is completed.