API Middlewares

Examples

API Routes with middleware

API Routes with CORS

API routes provide built in middlewares which parse the incoming request (`req`). Those middlewares are:

- `req.cookies` - An object containing the cookies sent by the request. Defaults to `{}`
- `req.query` - An object containing the query string. Defaults to `{}`
- `req.body` - An object containing the body parsed by `content-type`, or `null` if no body was sent

Custom config

Every API route can export a `config` object to change the default configs, which are the following:

```
export const config = {
  api: {
    bodyParser: {
      sizeLimit: '1mb',
    },
  },
}
```

The `api` object includes all configs available for API routes.

`bodyParser` is automatically enabled. If you want to consume the body as a `Stream` or with `raw-body`, you can set this to `false`.

One use case for disabling the automatic `bodyParsing` is to allow you to verify the raw body of a **webhook** request, for example from GitHub.

```
export const config = {
  api: {
    bodyParser: false,
  },
}
```

`bodyParser.sizeLimit` is the maximum size allowed for the parsed body, in any format supported by bytes, like so:

```
export const config = {
  api: {
    bodyParser: {
      sizeLimit: '500kb',
```

```
    },
  },
}
```

`externalResolver` is an explicit flag that tells the server that this route is being handled by an external resolver like *express* or *connect*. Enabling this option disables warnings for unresolved requests.

```
export const config = {
  api: {
    externalResolver: true,
  },
}
```

`responseLimit` is automatically enabled, warning when an API routes' response body is over 4MB.

If you are not using Next.js in a serverless environment, and understand the performance implications of not using a CDN or dedicated media host, you can set this limit to `false`.

```
export const config = {
  api: {
    responseLimit: false,
  },
}
```

`responseLimit` can also take the number of bytes or any string format supported by `bytes`, for example `1000`, `'500kb'` or `'3mb'`. This value will be the maximum response size before a warning is displayed. Default is 4MB. (see above)

```
export const config = {
  api: {
    responseLimit: '8mb',
  },
}
```

Connect/Express middleware support

You can also use Connect compatible middleware.

For example, configuring CORS for your API endpoint can be done leveraging the cors package.

First, install `cors`:

```
npm i cors
# or
yarn add cors
```

Now, let's add `cors` to the API route:

```
import Cors from 'cors'

// Initializing the cors middleware
const cors = Cors({
  methods: ['GET', 'HEAD'],
})

// Helper method to wait for a middleware to execute before continuing
// And to throw an error when an error happens in a middleware
function runMiddleware(req, res, fn) {
  return new Promise((resolve, reject) => {
    fn(req, res, (result) => {
      if (result instanceof Error) {
        return reject(result)
      }

      return resolve(result)
    })
  })
}

async function handler(req, res) {
  // Run the middleware
  await runMiddleware(req, res, cors)

  // Rest of the API logic
  res.json({ message: 'Hello Everyone!' })
}

export default handler
```

Go to the API Routes with CORS example to see the finished app

Extending the `req`/`res` objects with TypeScript

For better type-safety, it is not recommended to extend the `req` and `res` objects. Instead, use functions to work with them:

```
// utils/cookies.ts

import { serialize, CookieSerializeOptions } from 'cookie'
import { NextApiResponse } from 'next'

/**
 * This sets `cookie` using the `res` object
 */
```

```ts
export const setCookie = (
  res: NextApiResponse,
  name: string,
  value: unknown,
  options: CookieSerializeOptions = {}
) => {
  const stringValue =
    typeof value === 'object' ? 'j:' + JSON.stringify(value) : String(value)

  if ('maxAge' in options) {
    options.expires = new Date(Date.now() + options.maxAge)
    options.maxAge /= 1000
  }

  res.setHeader('Set-Cookie', serialize(name, stringValue, options))
}

// pages/api/cookies.ts

import { NextApiRequest, NextApiResponse } from 'next'
import { setCookie } from '../../utils/cookies'

const handler = (req: NextApiRequest, res: NextApiResponse) => {
  // Calling our pure function using the `res` object, it will add the `set-cookie` header
  setCookie(res, 'Next.js', 'api-middleware!')
  // Return the `set-cookie` header so we can display it in the browser and show that it wo
  res.end(res.getHeader('Set-Cookie'))
}

export default handler
```

If you can't avoid these objects from being extended, you have to create your own type to include the extra properties:

```ts
// pages/api/foo.ts

import { NextApiRequest, NextApiResponse } from 'next'
import { withFoo } from 'external-lib-foo'

type NextApiRequestWithFoo = NextApiRequest & {
  foo: (bar: string) => void
}

const handler = (req: NextApiRequestWithFoo, res: NextApiResponse) => {
  req.foo('bar') // we can now use `req.foo` without type errors
  res.end('ok')
}
```

```
export default withFoo(handler)
```

Keep in mind this is not safe since the code will still compile even if you remove `withFoo()` from the export.