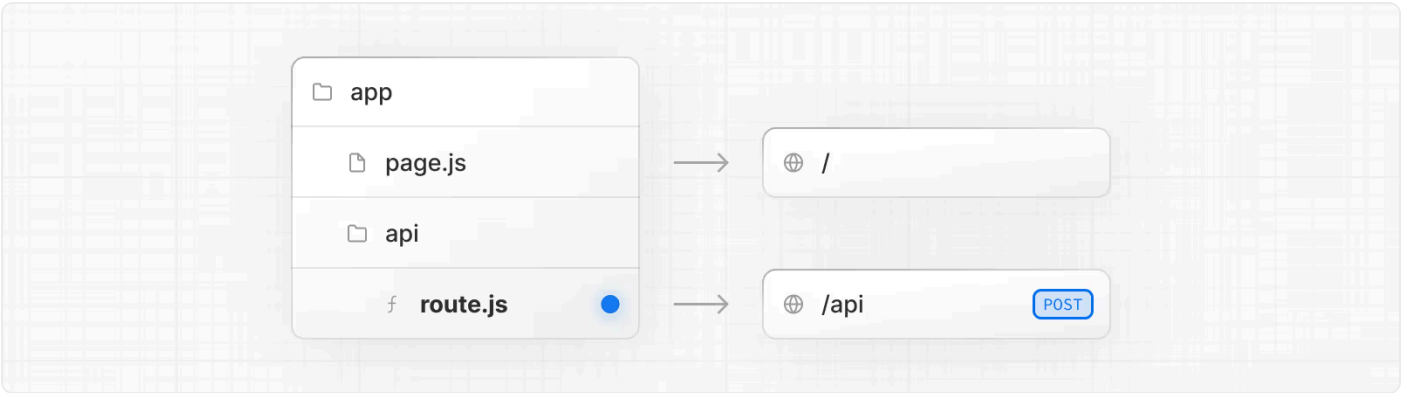
NEXT.JS

Menu

Building Your Application > Routing > Route Handlers

Route Handlers

Route Handlers allow you to create custom request handlers for a given route using the Web Request ↗ and Response ↗ APIs.



Good to know: Route Handlers are only available inside the `app` directory. They are the equivalent of API Routes inside the `pages` directory meaning you **do not** need to use API Routes and Route Handlers together.

Convention

Route Handlers are defined in a `route.js|ts` file inside the `app` directory:

app/api/route.ts TypeScript

```
export async function GET(request: Request) {}
```

Route Handlers can be nested anywhere inside the `app` directory, similar to `page.js` and `layout.js`. But there **cannot** be a `route.js` file at the same route segment level as `page.js`.

Supported HTTP Methods

The following HTTP methods ↗ are supported: `GET`, `POST`, `PUT`, `PATCH`, `DELETE`, `HEAD`, and `OPTIONS`. If an unsupported method is called, Next.js will return a `405 Method Not Allowed` response.

Extended `NextRequest` and `NextResponse` APIs

In addition to supporting the native Request ↗ and Response ↗ APIs, Next.js extends them with `NextRequest` and `NextResponse` to provide convenient helpers for advanced use cases.

Behavior

Caching

Route Handlers are not cached by default. You can, however, opt into caching for `GET` methods. Other supported HTTP methods are **not** cached. To cache a `GET` method, use a route config option such as `export const dynamic = 'force-static'` in your Route Handler file.

app/items/route.ts TypeScript

```ts
export const dynamic = 'force-static'

export async function GET() {
  const res = await fetch('https://data.mongodb-api.com/...', {
    headers: {
      'Content-Type': 'application/json',
      'API-Key': process.env.DATA_API_KEY,
    },
  })
```

```
  const data = await res.json()

  return Response.json({ data })
}
```

Good to know: Other supported HTTP methods are **not** cached, even if they are placed alongside a `GET` method that is cached, in the same file.

Special Route Handlers

Special Route Handlers like `sitemap.ts`, `opengraph-image.tsx`, and `icon.tsx`, and other metadata files remain static by default unless they use Dynamic APIs or dynamic config options.

Route Resolution

You can consider a `route` the lowest level routing primitive.

- They **do not** participate in layouts or client-side navigations like `page`.
- There **cannot** be a `route.js` file at the same route as `page.js`.

Page	Route	Result
`app/page.js`	`app/route.js`	✕ Conflict
`app/page.js`	`app/api/route.js`	✓ Valid
`app/[user]/page.js`	`app/api/route.js`	✓ Valid

Each `route.js` or `page.js` file takes over all HTTP verbs for that route.

app/page.ts TypeScript

```
export default function Page() {
  return <h1>Hello, Next.js!</h1>
}
```

```

// ❌ Conflict
// `app/route.ts`
export async function POST(request: Request) {}
```

Examples

The following examples show how to combine Route Handlers with other Next.js APIs and features.

Revalidating Cached Data

You can revalidate cached data using Incremental Static Regeneration (ISR):

app/posts/route.ts TypeScript

```
export const revalidate = 60

export async function GET() {
  const data = await fetch('https://api.vercel.app/blog')
  const posts = await data.json()

  return Response.json(posts)
}
```

Cookies

You can read or set cookies with `cookies` from `next/headers`. This server function can be called directly in a Route Handler, or nested inside of another function.

Alternatively, you can return a new `Response` using the `Set-Cookie` header.

app/api/route.ts TypeScript

```
import { cookies } from 'next/headers'
```

```
export async function GET(request: Request) {
  const cookieStore = await cookies()
  const token = cookieStore.get('token')

  return new Response('Hello, Next.js!', {
    status: 200,
    headers: { 'Set-Cookie': `token=${token.value}` },
  })
}
```

You can also use the underlying Web APIs to read cookies from the request (`NextRequest`):

app/api/route.ts TypeScript

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

export async function GET(request: NextRequest) {
  const token = request.cookies.get('token')
}
```

Headers

You can read headers with `headers` from `next/headers`. This server function can be called directly in a Route Handler, or nested inside of another function.

This `headers` instance is read-only. To set headers, you need to return a new `Response` with new `headers`.

app/api/route.ts TypeScript

```
import { headers } from 'next/headers'

export async function GET(request: Request) {
  const headersList = await headers()
  const referer = headersList.get('referer')

  return new Response('Hello, Next.js!', {
    status: 200,
    headers: { referer: referer },
  })
```

```
}
```

You can also use the underlying Web APIs to read headers from the request (`NextRequest`):

app/api/route.ts TypeScript

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

export async function GET(request: NextRequest) {
  const requestHeaders = new Headers(request.headers)
}
```

Redirects

app/api/route.ts TypeScript

```ts
import { redirect } from 'next/navigation'

export async function GET(request: Request) {
  redirect('https://nextjs.org/')
}
```

Dynamic Route Segments

Route Handlers can use Dynamic Segments to create request handlers from dynamic data.

app/items/[slug]/route.ts TypeScript

```ts
export async function GET(
  request: Request,
  { params }: { params: Promise<{ slug: string }> }
) {
  const slug = (await params).slug // 'a', 'b', or 'c'
}
```

Route	Example URL	`params`
`app/items/[slug]/route.js`	`/items/a`	`Promise<{ slug: 'a' }>`
`app/items/[slug]/route.js`	`/items/b`	`Promise<{ slug: 'b' }>`
`app/items/[slug]/route.js`	`/items/c`	`Promise<{ slug: 'c' }>`

URL Query Parameters

The request object passed to the Route Handler is a `NextRequest` instance, which has some additional convenience methods, including for more easily handling query parameters.

app/api/search/route.ts TypeScript

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

export function GET(request: NextRequest) {
  const searchParams = request.nextUrl.searchParams
  const query = searchParams.get('query')
  // query is "hello" for /api/search?query=hello
}
```

Streaming

Streaming is commonly used in combination with Large Language Models (LLMs), such as OpenAI, for AI-generated content. Learn more about the AI SDK ↗.

app/api/chat/route.ts TypeScript

```typescript
import { openai } from '@ai-sdk/openai'
import { StreamingTextResponse, streamText } from 'ai'

export async function POST(req: Request) {
  const { messages } = await req.json()
  const result = await streamText({
    model: openai('gpt-4-turbo'),
    messages,
```

```
  })

  return new StreamingTextResponse(result.toAIStream())
}
```

These abstractions use the Web APIs to create a stream. You can also use the underlying Web APIs directly.

app/api/route.ts TypeScript

```
// https://developer.mozilla.org/docs/Web/API/ReadableStream#convert_async_iterato
function iteratorToStream(iterator: any) {
  return new ReadableStream({
    async pull(controller) {
      const { value, done } = await iterator.next()

      if (done) {
        controller.close()
      } else {
        controller.enqueue(value)
      }
    },
  })
}

function sleep(time: number) {
  return new Promise((resolve) => {
    setTimeout(resolve, time)
  })
}

const encoder = new TextEncoder()

async function* makeIterator() {
  yield encoder.encode('<p>One</p>')
  await sleep(200)
  yield encoder.encode('<p>Two</p>')
  await sleep(200)
  yield encoder.encode('<p>Three</p>')
}

export async function GET() {
  const iterator = makeIterator()
  const stream = iteratorToStream(iterator)

```

```
  return new Response(stream)
}
```

Request Body

You can read the `Request` body using the standard Web API methods:

app/items/route.ts TypeScript

```
export async function POST(request: Request) {
  const res = await request.json()
  return Response.json({ res })
}
```

Request Body FormData

You can read the `FormData` using the `request.formData()` function:

app/items/route.ts TypeScript

```
export async function POST(request: Request) {
  const formData = await request.formData()
  const name = formData.get('name')
  const email = formData.get('email')
  return Response.json({ name, email })
}
```

Since `formData` data are all strings, you may want to use `zod-form-data` ↗ to validate the request and retrieve data in the format you prefer (e.g. `number`).

CORS

You can set CORS headers for a specific Route Handler using the standard Web API methods:

app/api/route.ts TypeScript

```
export async function GET(request: Request) {
```

```
  return new Response('Hello, Next.js!', {
    status: 200,
    headers: {
      'Access-Control-Allow-Origin': '*',
      'Access-Control-Allow-Methods': 'GET, POST, PUT, DELETE, OPTIONS',
      'Access-Control-Allow-Headers': 'Content-Type, Authorization',
    },
  })
}
```

Good to know:

- To add CORS headers to multiple Route Handlers, you can use Middleware or the `next.config.js` file.
- Alternatively, see our CORS example package.

Webhooks

You can use a Route Handler to receive webhooks from third-party services:

app/api/route.ts TypeScript

```
export async function POST(request: Request) {
  try {
    const text = await request.text()
    // Process the webhook payload
  } catch (error) {
    return new Response(`Webhook error: ${error.message}`, {
      status: 400,
    })
  }

  return new Response('Success!', {
    status: 200,
  })
}
```

Notably, unlike API Routes with the Pages Router, you do not need to use `bodyParser` to use any additional configuration.

Non-UI Responses

You can use Route Handlers to return non-UI content. Note that `sitemap.xml`, `robots.txt`, `app icons`, and open graph images all have built-in support.

app/rss.xml/route.ts TypeScript

```typescript
export async function GET() {
  return new Response(
    `<?xml version="1.0" encoding="UTF-8" ?>
<rss version="2.0">

<channel>
  <title>Next.js Documentation</title>
  <link>https://nextjs.org/docs</link>
  <description>The React Framework for the Web</description>
</channel>

</rss>`,
    {
      headers: {
        'Content-Type': 'text/xml',
      },
    }
  )
}
```

Segment Config Options

Route Handlers use the same route segment configuration as pages and layouts.

app/items/route.ts TypeScript

```typescript
export const dynamic = 'auto'
export const dynamicParams = true
export const revalidate = false
export const fetchCache = 'auto'
export const runtime = 'nodejs'
export const preferredRegion = 'auto'
```

See the API reference for more details.

API Reference

Learn more about the route.js file.

route.js

API reference for the route.js special file.

Previous
Intercepting Routes

Next
Middleware