API Routes

▶ **Examples**

Note: If you are using the App Router, you can use Server Components or Route Handlers instead of API Routes.

API routes provide a solution to build your **API** with Next.js.

Any file inside the folder `pages/api` is mapped to `/api/*` and will be treated as an API endpoint instead of a `page`. They are server-side only bundles and won't increase your client-side bundle size.

For example, the following API route `pages/api/user.js` returns a `json` response with a status code of `200`:

```
export default function handler(req, res) {
  res.status(200).json({ name: 'John Doe' });
}
```

Note: API Routes will be affected by `pageExtensions` configuration in `next.config.js`.

For an API route to work, you need to export a function as default (a.k.a **request handler**), which then receives the following parameters:

- `req`: An instance of http.IncomingMessage ↗, plus some pre-built middlewares
- `res`: An instance of http.ServerResponse ↗, plus some helper functions

To handle different HTTP methods in an API route, you can use `req.method` in your request handler, like so:

```
export default function handler(req, res) {
  if (req.method === 'POST') {
    // Process a POST request
  } else {
    // Handle any other HTTP method
  }
}
```

To fetch API endpoints, take a look into any of the examples at the start of this section.

Use Cases

For new projects, you can build your entire API with API Routes. If you have an existing API, you do not need to forward calls to the API through an API Route. Some other use cases for API Routes are:

- Masking the URL of an external service (e.g. `/api/secret` instead of `https://company.com/secret-url`)
- Using Environment Variables on the server to securely access external services.

Caveats

- API Routes do not specify CORS headers↗, meaning they are **same-origin only** by default. You can customize such behavior by wrapping the request handler with the CORS request helpers↗.
- API Routes can't be used with `output: 'export'`

Request Helpers

API Routes provide built-in request helpers which parse the incoming request (`req`):

- `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 configuration, which is the following:

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

The `api` object includes all config options 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',
  },
};
```

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

```ts
import { serialize, CookieSerializeOptions } from 'cookie';
import { NextApiResponse } from 'next';

/**
 * This sets `cookie` using the `res` object
 */
export const setCookie = (
  res: NextApiResponse,
  name: string,
  value: unknown,
  options: CookieSerializeOptions = {},
) => {
  const stringValue =
    typeof value === 'object' ? 'j:' + JSON.stringify(value) : String(value);

  if (typeof options.maxAge === 'number') {
    options.expires = new Date(Date.now() + options.maxAge * 1000);
  }

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

pages/api/cookies.ts

```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` heade
  // Add the `set-cookie` header on the main domain and expire after 30 days
  setCookie(res, 'Next.js', 'api-middleware!', { path: '/', maxAge: 2592000 });
  // Return the `set-cookie` header so we can display it in the browser and show that it
  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:

pages/api/foo.ts

```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.

Response Helpers

The Server Response object, (often abbreviated as `res`) includes a set of Express.js-like helper methods to improve the developer experience and increase the speed of creating new API endpoints.

The included helpers are:

- `res.status(code)` - A function to set the status code. `code` must be a valid HTTP status code
- `res.json(body)` - Sends a JSON response. `body` must be a serializable object
- `res.send(body)` - Sends the HTTP response. `body` can be a `string`, an `object` or a `Buffer`
- `res.redirect([status,] path)` - Redirects to a specified path or URL. `status` must be a valid HTTP status code. If not specified, `status` defaults to "307" "Temporary redirect".
- `res.revalidate(urlPath)` - Revalidate a page on demand using `getStaticProps`. `urlPath` must be a `string`.

Setting the status code of a response

When sending a response back to the client, you can set the status code of the response.

The following example sets the status code of the response to `200` (`OK`) and returns a `message` property with the value of `Hello from Next.js!` as a JSON response:

```
export default function handler(req, res) {
  res.status(200).json({ message: 'Hello from Next.js!' });
}
```

Sending a JSON response

When sending a response back to the client you can send a JSON response, this must be a serializable object. In a real world application you might want to let the client know the status of the request depending on the result of the requested endpoint.

The following example sends a JSON response with the status code `200` (`OK`) and the result of the async operation. It's contained in a try catch block to handle any errors that may occur, with the appropriate status code and error message caught and sent back to the client:

```
export default async function handler(req, res) {
  try {
    const result = await someAsyncOperation();
    res.status(200).json({ result });
  } catch (err) {
    res.status(500).json({ error: 'failed to load data' });
  }
}
```

Sending a HTTP response

Sending an HTTP response works the same way as when sending a JSON response. The only difference is that the response body can be a `string`, an `object` or a `Buffer`.

The following example sends a HTTP response with the status code `200` (`OK`) and the result of the async operation.

```
export default async function handler(req, res) {
  try {
    const result = await someAsyncOperation();
    res.status(200).send({ result });
  } catch (err) {
    res.status(500).send({ error: 'failed to fetch data' });
  }
}
```

Redirects to a specified path or URL

Taking a form as an example, you may want to redirect your client to a specified path or URL once they have submitted the form.

The following example redirects the client to the `/` path if the form is successfully submitted:

```
export default async function handler(req, res) {
  const { name, message } = req.body;
  try {
    await handleFormInputAsync({ name, message });
    res.redirect(307, '/');
  } catch (err) {
    res.status(500).send({ error: 'failed to fetch data' });
  }
}
```

Adding TypeScript types

You can make your response handlers more type-safe by importing the `NextApiRequest` and `NextApiResponse` types from `next`, in addition to those, you can also type your response data:

```
import type { NextApiRequest, NextApiResponse } from 'next';

type ResponseData = {
  message: string;
};

export default function handler(
  req: NextApiRequest,
  res: NextApiResponse<ResponseData>,
) {
  res.status(200).json({ message: 'Hello from Next.js!' });
}
```

Note: The body of `NextApiRequest` is `any` because the client may include any payload. You should validate the type/shape of the body at runtime before using it.

Dynamic API Routes

API routes support dynamic routes, and follow the same file naming rules used for `pages`.

For example, the API route `pages/api/post/[pid].js` has the following code:

```
export default function handler(req, res) {
  const { pid } = req.query;
  res.end(`Post: ${pid}`);
}
```

Now, a request to `/api/post/abc` will respond with the text: `Post: abc`.

Index routes and Dynamic API routes

A very common RESTful pattern is to set up routes like this:

- `GET api/posts` - gets a list of posts, probably paginated
- `GET api/posts/12345` - gets post id 12345

We can model this in two ways:

- Option 1:
  - `/api/posts.js`
  - `/api/posts/[postId].js`
- Option 2:
  - `/api/posts/index.js`
  - `/api/posts/[postId].js`

Both are equivalent. A third option of only using `/api/posts/[postId].js` is not valid because Dynamic Routes (including Catch-all routes - see below) do not have an `undefined` state and `GET api/posts` will not match `/api/posts/[postId].js` under any circumstances.

Catch all API routes

API Routes can be extended to catch all paths by adding three dots (`...`) inside the brackets. For example:

- `pages/api/post/[...slug].js` matches `/api/post/a`, but also `/api/post/a/b`, `/api/post/a/b/c` and so on.

> **Note**: You can use names other than `slug`, such as: `[...param]`

Matched parameters will be sent as a query parameter (`slug` in the example) to the page, and it will always be an array, so, the path `/api/post/a` will have the following `query` object:

```
{ "slug": ["a"] }
```

And in the case of `/api/post/a/b`, and any other matching path, new parameters will be added to the array, like so:

```
{ "slug": ["a", "b"] }
```

An API route for `pages/api/post/[...slug].js` could look like this:

```
export default function handler(req, res) {
  const { slug } = req.query;
  res.end(`Post: ${slug.join(', ')}`);
}
```

Now, a request to `/api/post/a/b/c` will respond with the text: `Post: a, b, c`.

Optional catch all API routes

Catch all routes can be made optional by including the parameter in double brackets (`[[...slug]]`).

For example, `pages/api/post/[[...slug]].js` will match `/api/post`, `/api/post/a`, `/api/post/a/b`, and so on.

The main difference between catch all and optional catch all routes is that with optional, the route without the parameter is also matched (`/api/post` in the example above).

The `query` objects are as follows:

```
{ } // GET `/api/post` (empty object)
{ "slug": ["a"] } // `GET /api/post/a` (single-element array)
{ "slug": ["a", "b"] } // `GET /api/post/a/b` (multi-element array)
```

Caveats

- Predefined API routes take precedence over dynamic API routes, and dynamic API routes over catch all API routes. Take a look at the following examples:
  - `pages/api/post/create.js` - Will match `/api/post/create`
  - `pages/api/post/[pid].js` - Will match `/api/post/1`, `/api/post/abc`, etc. But not `/api/post/create`
  - `pages/api/post/[...slug].js` - Will match `/api/post/1/2`, `/api/post/a/b/c`, etc. But not `/api/post/create`, `/api/post/abc`

Edge API Routes

Edge API Routes enable you to build high performance APIs with Next.js. Using the Edge Runtime, they are often faster than Node.js-based API Routes. This performance improvement does come with constraints, like not having access to native Node.js APIs. Instead, Edge API Routes are built on standard Web APIs.

Any file inside the folder `pages/api` is mapped to `/api/*` and will be treated as an API endpoint instead of a page. They are server-side only bundles and won't increase your client-side bundle size.

Examples

Basic

```
export const config = {
  runtime: 'edge',
};

export default (req) => new Response('Hello world!');
```

JSON Response

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

export const config = {
  runtime: 'edge',
};

export default async function handler(req: NextRequest) {
  return new Response(
    JSON.stringify({
      name: 'Jim Halpert',
    }),
    {
      status: 200,
      headers: {
        'content-type': 'application/json',
      },
    },
  );
}
```

Cache-Control

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

export const config = {
  runtime: 'edge',
};

export default async function handler(req: NextRequest) {
  return new Response(
    JSON.stringify({
      name: 'Jim Halpert',
    }),
    {
      status: 200,
      headers: {
        'content-type': 'application/json',
        'cache-control': 'public, s-maxage=1200, stale-while-revalidate=600',
      },
    },
  );
}
```

Query Parameters

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

export const config = {
  runtime: 'edge',
};

export default async function handler(req: NextRequest) {
  const { searchParams } = new URL(req.url);
  const email = searchParams.get('email');
  return new Response(email);
}
```

Forwarding Headers

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

export const config = {
  runtime: 'edge',
};

export default async function handler(req: NextRequest) {
  const authorization = req.cookies.get('authorization')?.value;
  return fetch('https://backend-api.com/api/protected', {
    method: req.method,
    headers: {
      authorization,
    },
    redirect: 'manual',
  });
}
```

Configuring Regions (for deploying)

You may want to restrict your edge function to specific regions when deploying so that you can colocate near your data sources ensuring lower response times which can be achieved as shown.

Note: This configuration is available in `v12.3.2` of Next.js and up.

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

export const config = {
  regions: ['sfo1', 'iad1'], // defaults to 'all'
};

export default async function handler(req: NextRequest) {
  const myData = await getNearbyData();
  return NextResponse.json(myData);
}
```

Differences between API Routes

Edge API Routes use the Edge Runtime, whereas API Routes use the Node.js runtime.

Edge API Routes can stream responses from the server and run *after* cached files (e.g. HTML, CSS, JavaScript) have been accessed. Server-side streaming can help improve performance with faster Time To First Byte (TTFB)↗.

> **Note:** Using Edge Runtime with `getServerSideProps` does not give you access to the response object. If you need access to `res`, you should use the Node.js runtime by setting `runtime: 'nodejs'`.

View the supported APIs and unsupported APIs for the Edge Runtime.