OpenTelemetry

Note: This feature is experimental, you need to explicitly opt-in by providing `experimental.instrumentationHook = true;` in your `next.config.js`.

Observability is crucial for understanding and optimizing the behavior and performance of your Next.js app.

As applications become more complex, it becomes increasingly difficult to identify and diagnose issues that may arise. By leveraging observability tools, such as logging and metrics, developers can gain insights into their application's behavior and identify areas for optimization. With observability, developers can proactively address issues before they become major problems and provide a better user experience. Therefore, it is highly recommended to use observability in your Next.js applications to improve performance, optimize resources, and enhance user experience.

We recommend using OpenTelemetry for instrumenting your apps. It's a platform-agnostic way to instrument apps that allows you to change your observability provider without changing your code. Read Official OpenTelemetry docs ↗ for more information about OpenTelemetry and how it works.

This documentation uses terms like *Span*, *Trace* or *Exporter* throughout this doc, all of which can be found in the OpenTelemetry Observability Primer ↗.

Next.js supports OpenTelemetry instrumentation out of the box, which means that we already instrumented Next.js itself. When you enable OpenTelemetry we will automatically wrap all your code like `getStaticProps` in *spans* with helpful attributes.

Note: We currently support OpenTelemetry bindings only in serverless functions. We don't provide any for `edge` or client side code.

Getting Started

OpenTelemetry is extensible but setting it up properly can be quite verbose. That's why we prepared a package `@vercel/otel` that helps you get started quickly. It's not extensible and you should configure OpenTelemetry manually you need to customize your setup.

Using `@vercel/otel`

To get started, you must install `@vercel/otel`:

Terminal

```
npm install @vercel/otel
```

Next, create a custom `instrumentation.ts` file in the root of the project:

instrumentation.ts

```ts
import { registerOTel } from '@vercel/otel';

export function register() {
  registerOTel('next-app');
}
```

Note: We have created a basic with-opentelemetry ↗ example that you can use.

Manual OpenTelemetry configuration

If our wrapper `@vercel/otel` doesn't suit your needs, you can configure OpenTelemetry manually.

Firstly you need to install OpenTelemetry packages:

Terminal

```
npm install @opentelemetry/sdk-node @opentelemetry/resources @opentelemetry/semantic-conventi
```

Now you can initialize `NodeSDK` in your `instrumentation.ts`. OpenTelemetry APIs are not compatible with edge runtime, so you need to make sure that you are importing them only when `process.env.NEXT_RUNTIME === 'nodejs'`. We recommend creating a new file `instrumentation.node.ts` which you conditionally import only when using node:

instrumentation.ts

```ts
export async function register() {
  if (process.env.NEXT_RUNTIME === 'nodejs') {
    await import('./instrumentation.node.ts');
  }
}
```

instrumentation.node.ts

```ts
import { trace, context } from '@opentelemetry/api';
import { NodeSDK } from '@opentelemetry/sdk-node';
import { OTLPTraceExporter } from '@opentelemetry/exporter-trace-otlp-http';
import { Resource } from '@opentelemetry/resources';
import { SemanticResourceAttributes } from '@opentelemetry/semantic-conventions';
import { SimpleSpanProcessor } from '@opentelemetry/sdk-trace-node';

const sdk = new NodeSDK({
  resource: new Resource({
    [SemanticResourceAttributes.SERVICE_NAME]: 'next-app',
  }),
  spanProcessor: new SimpleSpanProcessor(new OTLPTraceExporter()),
});
sdk.start();
```

Doing this is equivalent to using `@vercel/otel`, but it's possible to modify and extend. For example, you could use `@opentelemetry/exporter-trace-otlp-grpc` instead of `@opentelemetry/exporter-trace-otlp-http` or you can specify more resource attributes.

Testing your instrumentation

You need an OpenTelemetry collector with a compatible backend to test OpenTelemetry traces locally. We recommend using our OpenTelemetry dev environment ↗.

If everything works well you should be able to see the root server span labeled as `GET /requested/pathname`. All other spans from that particular trace will be nested under it.

Next.js traces more spans than are emitted by default. To see more spans, you must set `NEXT_OTEL_VERBOSE=1`.

Deployment

Using OpenTelemetry Collector

When you are deploying with OpenTelemetry Collector, you can use `@vercel/otel`. It will work both on Vercel and when self-hosted.

Deploying on Vercel

We made sure that OpenTelemetry works out of the box on Vercel.

Follow Vercel documentation ↗ to connect your project to an observability provider.

Self-hosting

Deploying to other platforms is also straightforward. You will need to spin up your own OpenTelemetry Collector to receive and process the telemetry data from your Next.js app.

To do this, follow the OpenTelemetry Collector Getting Started guide ↗, which will walk you through setting up the collector and configuring it to receive data from your Next.js app.

Once you have your collector up and running, you can deploy your Next.js app to your chosen platform following their respective deployment guides.

Custom Exporters

We recommend using OpenTelemetry Collector. If that is not possible on your platform, you can use a custom OpenTelemetry exporter with manual OpenTelemetry configuration

---

Custom Spans

You can add a custom span with OpenTelemetry APIs ↗.

```
>_ Terminal
npm install @opentelemetry/api
```

The following example demonstrates a function that fetches GitHub stars and adds a custom `fetchGithubStars` span to track the fetch request's result:

```
1  import { trace } from '@opentelemetry/api';
2
3  export async function fetchGithubStars() {
4    return await trace
```

```
    .getTracer('nextjs-example')
    .startActiveSpan('fetchGithubStars', async (span) => {
      try {
        return await getValue();
      } finally {
        span.end();
      }
    });
}
```

The `register` function will execute before your code runs in a new environment. You can start creating new spans, and they should be correctly added to the exported trace.

Default Spans in Next.js

Next.js automatically instruments several spans for you to provide useful insights into your application's performance.

Attributes on spans follow OpenTelemetry semantic conventions ↗. We also add some custom attributes under the `next` namespace:

- `next.span_name` - duplicates span name
- `next.span_type` - each span type has a unique identifier
- `next.route` - The route pattern of the request (e.g., `/[param]/user`).
- `next.page`
  - This is an internal value used by an app router.
  - You can think about it as a route to a special file (like `page.ts`, `layout.ts`, `loading.ts` and others)
  - It can be used as a unique identifier only when paired with `next.route` because `/layout` can be used to identify both `/(groupA)/layout.ts` and `/(groupB)/layout.ts`

`[http.method] [next.route]`

- `next.span_type`: `BaseServer.handleRequest`

This span represents the root span for each incoming request to your Next.js application. It tracks the HTTP method, route, target, and status code of the request.

Attributes:

- Common HTTP attributes↗
  - `http.method`
  - `http.status_code`
- Server HTTP attributes↗
  - `http.route`
  - `http.target`
- `next.span_name`
- `next.span_type`
- `next.route`

`render route (app) [next.route]`

- `next.span_type`: `AppRender.getBodyResult`.

This span represents the process of rendering a route in the app router.

Attributes:

- `next.span_name`
- `next.span_type`
- `next.route`

`fetch [http.method] [http.url]`

- `next.span_type`: `AppRender.fetch`

This span represents the fetch request executed in your code.

Attributes:

- Common HTTP attributes↗
  - `http.method`
- Client HTTP attributes↗
  - `http.url`
  - `net.peer.name`
  - `net.peer.port` (only if specified)
- `next.span_name`

- `next.span_type`

`executing api route (app) [next.route]`

- `next.span_type`: `AppRouteRouteHandlers.runHandler`.

This span represents the execution of an API route handler in the app router.

Attributes:

- `next.span_name`
- `next.span_type`
- `next.route`

`getServerSideProps [next.route]`

- `next.span_type`: `Render.getServerSideProps`.

This span represents the execution of `getServerSideProps` for a specific route.

Attributes:

- `next.span_name`
- `next.span_type`
- `next.route`

`getStaticProps [next.route]`

- `next.span_type`: `Render.getStaticProps`.

This span represents the execution of `getStaticProps` for a specific route.

Attributes:

- `next.span_name`
- `next.span_type`
- `next.route`

`render route (pages) [next.route]`

- `next.span_type`: `Render.renderDocument`.

This span represents the process of rendering the document for a specific route.

Attributes:

- `next.span_name`
- `next.span_type`
- `next.route`

`generateMetadata [next.page]`

- `next.span_type`: `ResolveMetadata.generateMetadata`.

This span represents the process of generating metadata for a specific page (a single route can have multiple of these spans).

Attributes:

- `next.span_name`
- `next.span_type`
- `next.page`