Script Component

Examples

Script Component

Google Tag Manager

Google Analytics

Facebook Pixel

Clerk

Segment Analytics

Version History

Version	Changes
`v11.0.0`	`next/script` introduced.

The Next.js Script component, `next/script`, is an extension of the HTML `<script>` element. It enables developers to set the loading priority of third-party scripts anywhere in their application, outside `next/head`, saving developer time while improving loading performance.

```
import Script from 'next/script'

export default function Home() {
  return (
    <>
      <Script src="https://www.google-analytics.com/analytics.js" />
    </>
  )
}
```

Overview

Websites often use third-party scripts to include different types of functionality into their site, such as analytics, ads, customer support widgets, and consent management. However, this can introduce problems that impact both user and developer experience:

- Some third-party scripts are heavy on loading performance and can drag down the user experience, especially if they are render-blocking and delay any page content from loading
- Developers often struggle to decide where to place third-party scripts in an application to ensure optimal loading

The Script component makes it easier for developers to place a third-party script anywhere in their application while taking care of optimizing its loading strategy.

Usage

To add a third-party script to your application, import the `next/script` component:

```
import Script from 'next/script'
```

Strategy

With `next/script`, you decide when to load your third-party script by using the `strategy` property:

```
<Script src="https://connect.facebook.net/en_US/sdk.js" strategy="lazyOnload" />
```

There are three different loading strategies that can be used:

- `beforeInteractive`: Load before the page is interactive
- `afterInteractive`: (**default**) Load immediately after the page becomes interactive
- `lazyOnload`: Load during idle time
- `worker`: (experimental) Load in a web worker

beforeInteractive Scripts that load with the `beforeInteractive` strategy are injected into the initial HTML from the server and run before self-bundled JavaScript is executed. This strategy should be used for any critical scripts that need to be fetched and executed before the page is interactive.

```
<Script
  src="https://cdn.jsdelivr.net/npm/cookieconsent@3/build/cookieconsent.min.js"
  strategy="beforeInteractive"
/>
```

Examples of scripts that should be loaded as soon as possible with this strategy include:

- Bot detectors
- Cookie consent managers

afterInteractive Scripts that use the `afterInteractive` strategy are injected client-side and will run after Next.js hydrates the page. This strategy should be used for scripts that do not need to load as soon as possible and can be fetched and executed immediately after the page is interactive.

```
<Script
  strategy="afterInteractive"
  dangerouslySetInnerHTML={{
    __html: `
    (function(w,d,s,l,i){w[l]=w[l]||[];w[l].push({'gtm.start':
    new Date().getTime(),event:'gtm.js'});var f=d.getElementsByTagName(s)[0],
    j=d.createElement(s),dl=l!='dataLayer'?'&l='+l:'';j.async=true;j.src=
    'https://www.googletagmanager.com/gtm.js?id='+i+dl;f.parentNode.insertBefore(j,f);
    })(window,document,'script','dataLayer', 'GTM-XXXXXX');
  `,
  }}
/>
```

Examples of scripts that are good candidates to load immediately after the page becomes interactive include:

- Tag managers
- Analytics

lazyOnload Scripts that use the `lazyOnload` strategy are loaded late after all resources have been fetched and during idle time. This strategy should be used for background or low priority scripts that do not need to load before or immediately after a page becomes interactive.

```
<Script src="https://connect.facebook.net/en_US/sdk.js" strategy="lazyOnload" />
```

Examples of scripts that do not need to load immediately and can be lazy-loaded include:

- Chat support plugins
- Social media widgets

Off-loading Scripts To A Web Worker (experimental)

> **Note: The `worker` strategy is not yet stable and can cause unexpected issues in your application. Use with caution.**

Scripts that use the `worker` strategy are relocated and executed in a web worker with Partytown. This can improve the performance of your site by dedicating the main thread to the rest of your application code.

This strategy is still experimental and can only be used if the `nextScriptWorkers` flag is enabled in `next.config.js`:

```
module.exports = {
  experimental: {
    nextScriptWorkers: true,
  },
}
```

Then, run `next` (normally `npm run dev` or `yarn dev`) and Next.js will guide you through the installation of the required packages to finish the setup:

```
npm run dev
```

```
# You'll see instructions like these:
#
# Please install Partytown by running:
#
#         npm install @builder.io/partytown
#
# ...
```

Once setup is complete, defining `strategy="worker"` will automatically instantiate Partytown in your application and off-load the script to a web worker.

```
<Script src="https://example.com/analytics.js" strategy="worker" />
```

There are a number of trade-offs that need to be considered when loading a third-party script in a web worker. Please see Partytown's Trade-Offs documentation for more information.

Configuration Although the `worker` strategy does not require any additional configuration to work, Partytown supports the use of a config object to modify some of its settings, including enabling `debug` mode and forwarding events and triggers.

If you would like to add additional configuration options, you can include it within the `<Head />` component used in a custom `_document.js`:

```
import { Html, Head, Main, NextScript } from 'next/document'

export default function Document() {
  return (
    <Html>
      <Head>
        <script
          data-partytown-config
          dangerouslySetInnerHTML={{
            __html: `
              partytown = {
                lib: "/_next/static/~partytown/",
                debug: true
              };
            `,
          }}
        />
      </Head>
      <body>
        <Main />
        <NextScript />
      </body>
    </Html>
  )
}
```

In order to modify Partytown's configuration, the following conditions must be met:

1. The `data-partytown-config` attribute must be used in order to overwrite the default configuration used by Next.js
2. Unless you decide to save Partytown's library files in a separate directory, the `lib: "/_next/static/~partytown/"` property and value must be included in the configuration object in order to let Partytown know where Next.js stores the necessary static files.

   Note: If you are using an asset prefix and would like to modify Partytown's default configuration, you must include it as part of the `lib` path.

Take a look at Partytown's configuration options to see the full list of other properties that can be added.

Inline Scripts

Inline scripts, or scripts not loaded from an external file, are also supported by the Script component. They can be written by placing the JavaScript within curly braces:

```
<Script id="show-banner" strategy="lazyOnload">
  {`document.getElementById('banner').classList.remove('hidden')`}
</Script>
```

Or by using the `dangerouslySetInnerHTML` property:

```
<Script
  id="show-banner"
  dangerouslySetInnerHTML={{
    __html: `document.getElementById('banner').classList.remove('hidden')`,
  }}
/>
```

There are two limitations to be aware of when using the Script component for inline scripts:

- Only the `afterInteractive` and `lazyOnload` strategies can be used. The `beforeInteractive` loading strategy injects the contents of an external script into the initial HTML response. Inline scripts already do this, which is why **the `beforeInteractive` strategy cannot be used with inline scripts.**
- An `id` attribute must be defined in order for Next.js to track and optimize the script

Executing Code After Loading (`onLoad`)

Some third-party scripts require users to run JavaScript code after the script has finished loading in order to instantiate content or call a function. If you are loading a script with either `beforeInteractive` or `afterInteractive` as a loading strategy, you can execute code after it has loaded using the `onLoad` property:

```
import { useState } from 'react'
import Script from 'next/script'

export default function Home() {
  const [stripe, setStripe] = useState(null)

  return (
    <>
      <Script
        id="stripe-js"
        src="https://js.stripe.com/v3/"
        onLoad={() => {
          setStripe({ stripe: window.Stripe('pk_test_12345') })
        }}
      />
    </>
  )
}
```

Sometimes it is helpful to catch when a script fails to load. These errors can be handled with the `onError` property:

```
import Script from 'next/script'

export default function Home() {
  return (
    <>
      <Script
        id="will-fail"
        src="https://example.com/non-existant-script.js"
        onError={(e) => {
          console.error('Script failed to load', e)
        }}
      />
    </>
  )
}
```

Additional Attributes

There are many DOM attributes that can be assigned to a `<script>` element that are not used by the Script component, like `nonce` or custom data attributes. Including any additional attributes will automatically forward it to the final, optimized `<script>` element that is outputted to the page.

```
import Script from 'next/script'

export default function Home() {
  return (
    <>
      <Script
        src="https://www.google-analytics.com/analytics.js"
        id="analytics"
        nonce="XUENAJFW"
        data-test="analytics"
      />
    </>
  )
}
```