Going to Production

Before taking your Next.js application to production, here are some recommendations to ensure the best user experience.

In General

- Use caching wherever possible.
- Ensure your database and backend are deployed in the same region.
- Aim to ship the least amount of JavaScript possible.
- Defer loading heavy JavaScript bundles until needed.
- Ensure logging is set up.
- Ensure error handling is set up.
- Configure the 404 (Not Found) and 500 (Error) pages.
- Ensure you are measuring performance.
- Run Lighthouse↗ to check for performance, best practices, accessibility, and SEO. For best results, use a production build of Next.js and use incognito in your browser so results aren't affected by extensions.
- Review Supported Browsers and Features.
- Improve performance using:
  - `next/image` and Automatic Image Optimization
  - Automatic Font Optimization
  - Script Optimization
- Improve loading performance

Caching

▼ **Examples**

- ssr-caching ↗

Caching improves response times and reduces the number of requests to external services. Next.js automatically adds caching headers to immutable assets served from `/_next/static` including JavaScript, CSS, static images, and other media.

```
Cache-Control: public, max-age=31536000, immutable
```

`Cache-Control` headers set in `next.config.js` will be overwritten in production to ensure that static assets can be cached effectively. If you need to revalidate the cache of a page that has been statically generated, you can do so by setting `revalidate` in the page's `getStaticProps` function. If you're using `next/image`, you can configure the `minimumCacheTTL` for the default Image Optimization loader.

Note: When running your application locally with `next dev`, your headers are overwritten to prevent caching locally.

```
Cache-Control: no-cache, no-store, max-age=0, must-revalidate
```

You can also use caching headers inside `getServerSideProps` and API Routes for dynamic responses. For example, using `stale-while-revalidate` ↗.

```
// This value is considered fresh for ten seconds (s-maxage=10).
// If a request is repeated within the next 10 seconds, the previously
// cached value will still be fresh. If the request is repeated before 59 seconds,
// the cached value will be stale but still render (stale-while-revalidate=59).
//
// In the background, a revalidation request will be made to populate the cache
// with a fresh value. If you refresh the page, you will see the new value.
export async function getServerSideProps({ req, res }) {
  res.setHeader(
    'Cache-Control',
    'public, s-maxage=10, stale-while-revalidate=59',
  );

  return {
    props: {},
  };
}
```

By default, `Cache-Control` headers will be set differently depending on how your page fetches data.

- If the page uses `getServerSideProps` or `getInitialProps`, it will use the default `Cache-Control` header set by `next start` in order to prevent accidental caching of responses that cannot be cached. If you want a different cache behavior while using `getServerSideProps`, use `res.setHeader('Cache-Control', 'value_you_prefer')` inside of the function as shown above.
- If the page is using `getStaticProps`, it will have a `Cache-Control` header of `s-maxage=REVALIDATE_SECONDS, stale-while-revalidate`, or if `revalidate` is *not* used, `s-maxage=31536000, stale-while-revalidate` to cache for the maximum age possible.

Note: Your deployment provider must support caching for dynamic responses. If you are self-hosting, you will need to add this logic yourself using a key/value store like Redis. If you are using Vercel, Edge Caching works without configuration ↗.

Reducing JavaScript Size

▼ **Examples**

- with-dynamic-import ↗

To reduce the amount of JavaScript sent to the browser, you can use the following tools to understand what is included inside each JavaScript bundle:

- Import Cost ↗ – Display the size of the imported package inside VSCode.
- Package Phobia ↗ – Find the cost of adding a new dev dependency to your project.
- Bundle Phobia ↗ - Analyze how much a dependency can increase bundle sizes.
- Webpack Bundle Analyzer ↗ – Visualize the size of webpack output files with an interactive, zoomable treemap.
- bundlejs ↗ - An online tool to quickly bundle & minify your projects, while viewing the compressed gzip/brotli bundle size, all running locally on your browser.

Each file inside your `pages/` directory will automatically be code split into its own JavaScript bundle during `next build`. You can also use Dynamic Imports to lazy-load components and libraries. For example, you might want to defer loading your modal code until a user clicks the open button.

Logging

▼ **Examples**

- Pino and Logflare Example ↗

Since Next.js runs on both the client and server, there are multiple forms of logging supported:

- `console.log` in the browser
- `stdout` on the server

If you want a structured logging package, we recommend Pino ↗. If you're using Vercel, there are pre-built logging integrations ↗ compatible with Next.js.

Error Handling

▼ **Examples**

- with-sentry ↗

When an unhandled exception occurs, you can control the experience for your users with the 500 page. We recommend customizing this to your brand instead of the default Next.js theme.

You can also log and track exceptions with a tool like Sentry. This example ↗ shows how to catch & report errors on both the client and server-side, using the Sentry SDK for Next.js. There's also a Sentry integration for Vercel ↗.

Loading Performance

To improve loading performance, you first need to determine what to measure and how to measure it. Core Web Vitals ↗ is a good industry standard that is measured using your own web browser. If you are not familiar with the metrics of Core Web Vitals, review this blog post ↗ and determine which specific metric/s will be your drivers for loading performance. Ideally, you would want to measure the loading performance in the following environments:

- In the lab, using your own computer or a simulator.

- In the field, using real-world data from actual visitors.
- Local, using a test that runs on your device.
- Remote, using a test that runs in the cloud.

Once you are able to measure the loading performance, use the following strategies to improve it iteratively so that you apply one strategy, measure the new performance and continue tweaking until you do not see much improvement. Then, you can move on to the next strategy.

- Use caching regions that are close to the regions where your database or API is deployed.
- As described in the caching section, use a `stale-while-revalidate` value that will not overload your backend.
- Use Incremental Static Regeneration to reduce the number of requests to your backend.
- Remove unused JavaScript. Review this blog post ↗ to understand what Core Web Vitals metrics bundle size affects and what strategies you can use to reduce it, such as:
  - Setting up your Code Editor to view import costs and sizes
  - Finding alternative smaller packages
  - Dynamically loading components and dependencies