Incremental Static Regeneration

▶ **Examples**

▶ **Version History**

Next.js allows you to create or update static pages *after* you've built your site. Incremental Static Regeneration (ISR) enables you to use static-generation on a per-page basis, **without needing to rebuild the entire site**. With ISR, you can retain the benefits of static while scaling to millions of pages.

To use ISR, add the `revalidate` prop to `getStaticProps`:

```
function Blog({ posts }) {
  return (
    <ul>
      {posts.map((post) => (
        <li key={post.id}>{post.title}</li>
      ))}
    </ul>
  )
}

// This function gets called at build time on server-side.
// It may be called again, on a serverless function, if
// revalidation is enabled and a new request comes in
export async function getStaticProps() {
  const res = await fetch('https://.../posts')
  const posts = await res.json()

  return {
    props: {
      posts,
    },
    // Next.js will attempt to re-generate the page:
    // - When a request comes in
    // - At most once every 10 seconds
    revalidate: 10, // In seconds
  }
}

// This function gets called at build time on server-side.
// It may be called again, on a serverless function, if
// the path has not been generated.
export async function getStaticPaths() {
  const res = await fetch('https://.../posts')
  const posts = await res.json()

  // Get the paths we want to pre-render based on posts
  const paths = posts.map((post) => ({
    params: { id: post.id },
  }))

  // We'll pre-render only these paths at build time.
```

```
  // { fallback: blocking } will server-render pages
  // on-demand if the path doesn't exist.
  return { paths, fallback: 'blocking' }
}

export default Blog
```

When a request is made to a page that was pre-rendered at build time, it will initially show the cached page.

- Any requests to the page after the initial request and before 10 seconds are also cached and instantaneous.
- After the 10-second window, the next request will still show the cached (stale) page
- Next.js triggers a regeneration of the page in the background.
- Once the page generates successfully, Next.js will invalidate the cache and show the updated page. If the background regeneration fails, the old page would still be unaltered.

When a request is made to a path that hasn't been generated, Next.js will server-render the page on the first request. Future requests will serve the static file from the cache. ISR on Vercel persists the cache globally and handles rollbacks.

On-demand Revalidation (Beta)

If you set a `revalidate` time of `60`, all visitors will see the same generated version of your site for one minute. The only way to invalidate the cache is from someone visiting that page after the minute has passed.

Starting with `v12.1.0`, Next.js supports on-demand Incremental Static Regeneration to manually purge the Next.js cache for a specific page. This makes it easier to update your site when:

- Content from your headless CMS is created or updated
- Ecommerce metadata changes (price, description, category, reviews, etc.)

Inside `getStaticProps`, you do not need to specify `revalidate` to use on-demand revalidation. If `revalidate` is omitted, Next.js will use the default value of `false` (no revalidation) and only revalidate the page on-demand when `unstable_revalidate` is called.

Using On-Demand Revalidation

First, create a secret token only known by your Next.js app. This secret will be used to prevent unauthorized access to the revalidation API Route. You can access the route (either manually or with a webhook) with the following URL structure:

```
https://<your-site.com>/api/revalidate?secret=<token>
```

Next, add the secret as an Environment Variable to your application. Finally, create the revalidation API Route:

```
// pages/api/revalidate.js

export default async function handler(req, res) {
  // Check for secret to confirm this is a valid request
  if (req.query.secret !== process.env.MY_SECRET_TOKEN) {
    return res.status(401).json({ message: 'Invalid token' })
  }
```

```
  try {
    await res.unstable_revalidate('/path-to-revalidate')
    return res.json({ revalidated: true })
  } catch (err) {
    // If there was an error, Next.js will continue
    // to show the last successfully generated page
    return res.status(500).send('Error revalidating')
  }
}
```

View our demo to see on-demand revalidation in action and provide feedback.

Testing on-demand ISR during development

When running locally with `next dev`, `getStaticProps` is invoked on every request. To verify your on-demand ISR configuration is correct, you will need to create a production build and start the production server:

```
$ next build
$ next start
```

Then, you are able to validate static pages are successfully revalidated.

Error handling and revalidation

If there is an error inside `getStaticProps` when handling background regeneration, or you manually throw an error, the last successfully generated page will continue to show. On the next subsequent request, Next.js will retry calling `getStaticProps`.

```
export async function getStaticProps() {
  // If this request throws an uncaught error, Next.js will
  // not invalidate the currently shown page and
  // retry getStaticProps on the next request.
  const res = await fetch('https://.../posts')
  const posts = await res.json()

  if (!res.ok) {
    // If there is a server error, you might want to
    // throw an error instead of returning so that the cache is not updated
    // until the next successful request.
    throw new Error(`Failed to fetch posts, received status ${res.status}`)
  }

  // If the request was successful, return the posts
  // and revalidate every 10 seconds.
  return {
    props: {
      posts,
    },
    revalidate: 10,
  }
}
```

Related

For more information on what to do next, we recommend the following sections:

Dynamic routing Learn more about dynamic routing in Next.js with getStaticPaths.