<Image>

▼ **Examples**

- Image Component

This API reference will help you understand how to use props and configuration options available for the Image Component. For features and usage, please see the Image Component page.

app/page.js

```js
import Image from 'next/image'

export default function Page() {
  return (
    <Image
      src="/profile.png"
      width={500}
      height={500}
      alt="Picture of the author"
    />
  )
}
```

Props

Here's a summary of the props available for the Image Component:

Prop	Example	Type	Status
`src`	`src="/profile.png"`	String	Required
`width`	`width={500}`	Integer (px)	Required
`height`	`height={500}`	Integer (px)	Required
`alt`	`alt="Picture of the author"`	String	Required
`loader`	`loader={imageLoader}`	Function	-
`fill`	`fill={true}`	Boolean	-
`sizes`	`sizes="(max-width: 768px) 100vw, 33vw"`	String	-
`quality`	`quality={80}`	Integer (1-100)	-
`priority`	`priority={true}`	Boolean	-
`placeholder`	`placeholder="blur"`	String	-
`style`	`style={{objectFit: "contain"}}`	Object	-
`onLoadingComplete`	`onLoadingComplete={img => done())}`	Function	Deprecated
`onLoad`	`onLoad={event => done())}`	Function	-
`onError`	`onError(event => fail()}`	Function	-
`loading`	`loading="lazy"`	String	-
`blurDataURL`	`blurDataURL="data:image/jpeg..."`	String	-
`overrideSrc`	`overrideSrc="/seo.png"`	String	-

Required Props

The Image Component requires the following properties: `src`, `alt`, `width` and `height` (or `fill`).

app/page.js

```js
import Image from 'next/image'

export default function Page() {
  return (
    <div>
      <Image
        src="/profile.png"
        width={500}
        height={500}
        alt="Picture of the author"
      />
    </div>
  )
}
```

`src`

Must be one of the following:

- A statically imported image file
- A path string. This can be either an absolute external URL, or an internal path depending on the loader prop.

When using an external URL, you must add it to remotePatterns in `next.config.js`.

`width`

The `width` property represents the *intrinsic* image width in pixels.

Required, except for statically imported images or images with the `fill` property.

`height`

The `height` property represents the *intrinsic* image height in pixels.

Required, except for statically imported images or images with the `fill` property.

> **Good to know:**
>
> - Combined, both `width` and `height` properties are used to determine the aspect ratio of the image which used by browsers to reserve space for the image before it loads.
> - The intrinsic size does not always mean the rendered size in the browser, which will be determined by the parent container. For example, if the parent container is smaller than the intrinsic size, the image will be scaled down to fit the container.
> - You can use the `fill` property when the width and height are unknown.

`alt`

The `alt` property is used to describe the image for screen readers and search engines. It is also the fallback text if images have been disabled or an error occurs while loading the image.

It should contain text that could replace the image without changing the meaning of the page ↗. It is not meant to supplement the image and should not repeat information that is already provided in the captions above or below the image.

If the image is purely decorative ↗ or not intended for the user ↗, the `alt` property should be an empty string (`alt=""`).

Learn more ↗

Optional Props

The `<Image />` component accepts a number of additional properties beyond those which are required. This section describes the most commonly-used properties of the Image component. Find details about more rarely-used properties in the Advanced Props section.

`loader`

A custom function used to resolve image URLs.

A `loader` is a function returning a URL string for the image, given the following parameters:

- `src`
- `width`
- `quality`

Here is an example of using a custom loader:

```
'use client'

import Image from 'next/image'

const imageLoader = ({ src, width, quality }) => {
  return `https://example.com/${src}?w=${width}&q=${quality || 75}`
}

export default function Page() {
  return (
    <Image
      loader={imageLoader}
      src="me.png"
      alt="Picture of the author"
      width={500}
      height={500}
    />
  )
}
```

Good to know: Using props like `loader`, which accept a function, requires using Client Components to serialize the provided function.

Alternatively, you can use the loaderFile configuration in `next.config.js` to configure every instance of `next/image` in your application, without passing a prop.

`fill`

```
fill={true} // {true} | {false}
```

A boolean that causes the image to fill the parent element, which is useful when the `width` and `height` are unknown.

The parent element *must* assign `position: "relative"`, `position: "fixed"`, or `position: "absolute"` style.

By default, the img element will automatically be assigned the `position: "absolute"` style.

If no styles are applied to the image, the image will stretch to fit the container. You may prefer to set `object-fit: "contain"` for an image which is letterboxed to fit the container and preserve aspect ratio.

Alternatively, `object-fit: "cover"` will cause the image to fill the entire container and be cropped to preserve aspect ratio.

For more information, see also:

- `position` ↗
- `object-fit` ↗
- `object-position` ↗

`sizes`

A string, similar to a media query, that provides information about how wide the image will be at different breakpoints. The value of `sizes` will greatly affect performance for images using `fill` or which are styled to have a responsive size.

The `sizes` property serves two important purposes related to image performance:

- First, the value of `sizes` is used by the browser to determine which size of the image to download, from `next/image`'s automatically generated `srcset`. When the browser chooses, it does not yet know the size of the image on the page, so it selects an image that is the same size or larger than the viewport. The `sizes` property allows you to tell the browser that the image will actually be smaller than full screen. If you don't specify a `sizes` value in an image with the `fill` property, a default value of `100vw` (full screen width) is used.

- Second, the `sizes` property changes the behavior of the automatically generated `srcset` value. If no `sizes` value is present, a small `srcset` is generated, suitable for a fixed-size image (1x/2x/etc). If `sizes` is defined, a large `srcset` is generated, suitable for a responsive image (640w/750w/etc). If the `sizes` property includes sizes such as `50vw`, which represent a percentage of the viewport width, then the `srcset` is trimmed to not include any values which are too small to ever be necessary.

For example, if you know your styling will cause an image to be full-width on mobile devices, in a 2-column layout on tablets, and a 3-column layout on desktop displays, you should include a sizes property such as the following:

```
import Image from 'next/image'

export default function Page() {
  return (
    <div className="grid-element">
      <Image
        fill
        src="/example.png"
        sizes="(max-width: 768px) 100vw, (max-width: 1200px) 50vw, 33vw"
      />
    </div>
  )
}
```

This example `sizes` could have a dramatic effect on performance metrics. Without the `33vw` sizes, the image selected from the server would be 3 times as wide as it needs to be. Because file size is proportional to the square of the width, without `sizes` the user would download an image that's 9 times larger than necessary.

Learn more about `srcset` and `sizes`:

- web.dev ↗
- mdn ↗

`quality`

```
quality={75} // {number 1-100}
```

The quality of the optimized image, an integer between `1` and `100`, where `100` is the best quality and therefore largest file size. Defaults to `75`.

`priority`

```
priority={false} // {false} | {true}
```

When true, the image will be considered high priority and preload↗. Lazy loading is automatically disabled for images using `priority`.

You should use the `priority` property on any image detected as the Largest Contentful Paint (LCP)↗ element. It may be appropriate to have multiple priority images, as different images may be the LCP element for different viewport sizes.

Should only be used when the image is visible above the fold. Defaults to `false`.

`placeholder`

```
placeholder = 'empty' // "empty" | "blur" | "data:image/..."
```

A placeholder to use while the image is loading. Possible values are `blur`, `empty`, or `data:image/...`. Defaults to `empty`.

When `blur`, the `blurDataURL` property will be used as the placeholder. If `src` is an object from a static import and the imported image is `.jpg`, `.png`, `.webp`, or `.avif`, then `blurDataURL` will be automatically populated, except when the image is detected to be animated.

For dynamic images, you must provide the `blurDataURL` property. Solutions such as Plaiceholder↗ can help with `base64` generation.

When `data:image/...`, the Data URL ↗ will be used as the placeholder while the image is loading.

When `empty`, there will be no placeholder while the image is loading, only empty space.

Try it out:

- Demo the `blur` placeholder ↗
- Demo the shimmer effect with data URL `placeholder` prop ↗
- Demo the color effect with `blurDataURL` prop ↗

Advanced Props

In some cases, you may need more advanced usage. The `<Image />` component optionally accepts the following advanced properties.

`style`

Allows passing CSS styles to the underlying image element.

components/ProfileImage.js

```
const imageStyle = {
  borderRadius: '50%',
  border: '1px solid #fff',
}

export default function ProfileImage() {
  return <Image src="..." style={imageStyle} />
}
```

Remember that the required width and height props can interact with your styling. If you use styling to modify an image's width, you should also style its height to `auto` to preserve its intrinsic aspect ratio, or your image will be distorted.

`onLoadingComplete`

```
'use client'

<Image onLoadingComplete={(img) => console.log(img.naturalWidth)} />
```

> **Warning**: Deprecated since Next.js 14 in favor of `onLoad`.

A callback function that is invoked once the image is completely loaded and the placeholder has been removed.

The callback function will be called with one argument, a reference to the underlying `<img>` element.

> **Good to know**: Using props like `onLoadingComplete`, which accept a function, requires using Client Components to serialize the provided function.

`onLoad`

```
<Image onLoad={(e) => console.log(e.target.naturalWidth)} />
```

A callback function that is invoked once the image is completely loaded and the placeholder has been removed.

The callback function will be called with one argument, the Event which has a `target` that references the underlying `<img>` element.

> **Good to know**: Using props like `onLoad`, which accept a function, requires using Client Components to serialize the provided function.

`onError`

```
<Image onError={(e) => console.error(e.target.id)} />
```

A callback function that is invoked if the image fails to load.

> **Good to know**: Using props like `onError`, which accept a function, requires using Client Components to serialize the provided function.

`loading`

> **Recommendation**: This property is only meant for advanced use cases. Switching an image to load with `eager` will normally **hurt performance**. We recommend using the `priority` property instead, which will eagerly preload the image.

```
loading = 'lazy' // {lazy} | {eager}
```

The loading behavior of the image. Defaults to `lazy`.

When `lazy`, defer loading the image until it reaches a calculated distance from the viewport.

When `eager`, load the image immediately.

Learn more about the `loading` attribute↗.

`blurDataURL`

A Data URL↗ to be used as a placeholder image before the `src` image successfully loads. Only takes effect when combined with `placeholder="blur"`.

Must be a base64-encoded image. It will be enlarged and blurred, so a very small image (10px or less) is recommended. Including larger images as placeholders may harm your application performance.

Try it out:

- Demo the default `blurDataURL` prop↗

- Demo the color effect with `blurDataURL` prop ↗

You can also generate a solid color Data URL ↗ to match the image.

`unoptimized`

```
unoptimized = {false} // {false} | {true}
```

When true, the source image will be served as-is instead of changing quality, size, or format. Defaults to `false`.

```
import Image from 'next/image'

const UnoptimizedImage = (props) => {
  return <Image {...props} unoptimized />
}
```

Since Next.js 12.3.0, this prop can be assigned to all images by updating `next.config.js` with the following configuration:

next.config.js

```js
module.exports = {
  images: {
    unoptimized: true,
  },
}
```

`overrideSrc`

When providing the `src` prop to the `<Image>` component, both the `srcset` and `src` attributes are generated automatically for the resulting `<img>`.

input.js

```jsx
<Image src="/me.jpg" />
```

output.html

```html
<img
  srcset="
    /_next/image?url=%2Fme.jpg&w=640&q=75 1x,
    /_next/image?url=%2Fme.jpg&w=828&q=75 2x
  "
  src="/_next/image?url=%2Fme.jpg&w=828&q=75"
/>
```

In some cases, it is not desirable to have the `src` attribute generated and you may wish to override it using the `overrideSrc` prop.

For example, when upgrading an existing website from `<img>` to `<Image>`, you may wish to maintain the same `src` attribute for SEO purposes such as image ranking or avoiding recrawl.

input.js

```jsx
<Image src="/me.jpg" overrideSrc="/override.jpg" />
```

output.html

```html
<img
  srcset="
    /_next/image?url=%2Fme.jpg&w=640&q=75 1x,
    /_next/image?url=%2Fme.jpg&w=828&q=75 2x
  "
  src="/override.jpg"
/>
```

Other Props

Other properties on the `<Image />` component will be passed to the underlying `img` element with the exception of the following:

- `srcSet`. Use Device Sizes instead.
- `decoding`. It is always `"async"`.

Configuration Options

In addition to props, you can configure the Image Component in `next.config.js`. The following options are available:

`remotePatterns`

To protect your application from malicious users, configuration is required in order to use external images. This ensures that only external images from your account can be served from the Next.js Image Optimization API. These external images can be configured with the `remotePatterns` property in your `next.config.js` file, as shown below:

next.config.js

```js
module.exports = {
  images: {
    remotePatterns: [
      {
        protocol: 'https',
        hostname: 'example.com',
        port: '',
        pathname: '/account123/**',
      },
    ],
  },
}
```

Good to know: The example above will ensure the `src` property of `next/image` must start with `https://example.com/account123/`. Any other protocol, hostname, port, or unmatched path will respond with 400 Bad Request.

Below is another example of the `remotePatterns` property in the `next.config.js` file:

next.config.js

```js
module.exports = {
  images: {
    remotePatterns: [
      {
        protocol: 'https',
        hostname: '**.example.com',
        port: '',
      },
    ],
  },
}
```

> **Good to know**: The example above will ensure the `src` property of `next/image` must start with `https://img1.example.com` or `https://me.avatar.example.com` or any number of subdomains. Any other protocol, port, or unmatched hostname will respond with 400 Bad Request.

Wildcard patterns can be used for both `pathname` and `hostname` and have the following syntax:

- `*` match a single path segment or subdomain
- `**` match any number of path segments at the end or subdomains at the beginning

The `**` syntax does not work in the middle of the pattern.

> **Good to know**: When omitting `protocol`, `port` or `pathname`, then the wildcard `**` is implied. This is not recommended because it may allow malicious actors to optimize urls you did not intend.

`domains`

> **Warning**: Deprecated since Next.js 14 in favor of strict `remotePatterns` in order to protect your application from malicious users. Only use `domains` if you own all the content served from the domain.

Similar to `remotePatterns`, the `domains` configuration can be used to provide a list of allowed hostnames for external images.

However, the `domains` configuration does not support wildcard pattern matching and it cannot restrict protocol, port, or pathname.

Below is an example of the `domains` property in the `next.config.js` file:

next.config.js

```
module.exports = {
  images: {
    domains: ['assets.acme.com'],
  },
}
```

`loaderFile`

If you want to use a cloud provider to optimize images instead of using the Next.js built-in Image Optimization API, you can configure the `loaderFile` in your `next.config.js` like the following:

next.config.js

```
module.exports = {
  images: {
    loader: 'custom',
    loaderFile: './my/image/loader.js',
  },
}
```

This must point to a file relative to the root of your Next.js application. The file must export a default function that returns a string, for example:

my/image/loader.js

```
'use client'

export default function myImageLoader({ src, width, quality }) {
  return `https://example.com/${src}?w=${width}&q=${quality || 75}`
}
```

Alternatively, you can use the `loader` prop to configure each instance of `next/image`.

Examples:

- Custom Image Loader Configuration

> **Good to know**: Customizing the image loader file, which accepts a function, requires using Client Components to serialize the provided function.

Advanced

The following configuration is for advanced use cases and is usually not necessary. If you choose to configure the properties below, you will override any changes to the Next.js defaults in future updates.

`deviceSizes`

If you know the expected device widths of your users, you can specify a list of device width breakpoints using the `deviceSizes` property in `next.config.js`. These widths are used when the `next/image` component uses `sizes` prop to ensure the correct image is served for user's device.

If no configuration is provided, the default below is used.

next.config.js

```
module.exports = {
  images: {
    deviceSizes: [640, 750, 828, 1080, 1200, 1920, 2048, 3840],
  },
}
```

`imageSizes`

You can specify a list of image widths using the `images.imageSizes` property in your `next.config.js` file. These widths are concatenated with the array of device sizes to form the full array of sizes used to generate image srcset↗s.

The reason there are two separate lists is that imageSizes is only used for images which provide a `sizes` prop, which indicates that the image is less than the full width of the screen. **Therefore, the sizes in imageSizes should all be smaller than the smallest size in deviceSizes.**

If no configuration is provided, the default below is used.

next.config.js

```
module.exports = {
  images: {
    imageSizes: [16, 32, 48, 64, 96, 128, 256, 384],
  },
}
```

`formats`

The default Image Optimization API will automatically detect the browser's supported image formats via the request's `Accept` header.

If the `Accept` head matches more than one of the configured formats, the first match in the array is used. Therefore, the array order matters. If there is no match (or the source image is animated), the Image Optimization API will fallback to the original image's format.

If no configuration is provided, the default below is used.

next.config.js

```js
module.exports = {
  images: {
    formats: ['image/webp'],
  },
}
```

You can enable AVIF support with the following configuration.

next.config.js

```js
module.exports = {
  images: {
    formats: ['image/avif', 'image/webp'],
  },
}
```

> **Good to know**:
>
> - AVIF generally takes 20% longer to encode but it compresses 20% smaller compared to WebP. This means that the first time an image is requested, it will typically be slower and then subsequent requests that are cached will be faster.
> - If you self-host with a Proxy/CDN in front of Next.js, you must configure the Proxy to forward the `Accept` header.

Caching Behavior

The following describes the caching algorithm for the default loader. For all other loaders, please refer to your cloud provider's documentation.

Images are optimized dynamically upon request and stored in the `<distDir>/cache/images` directory. The optimized image file will be served for subsequent requests until the expiration is reached. When a request is made that matches a cached but expired file, the expired image is served stale immediately. Then the image is optimized again in the background (also called revalidation) and saved to the cache with the new expiration date.

The cache status of an image can be determined by reading the value of the `x-nextjs-cache` response header. The possible values are the following:

- `MISS` - the path is not in the cache (occurs at most once, on the first visit)
- `STALE` - the path is in the cache but exceeded the revalidate time so it will be updated in the background
- `HIT` - the path is in the cache and has not exceeded the revalidate time

The expiration (or rather Max Age) is defined by either the `minimumCacheTTL` configuration or the upstream image `Cache-Control` header, whichever is larger. Specifically, the `max-age` value of the `Cache-Control` header is used. If both `s-maxage` and `max-age` are found, then `s-maxage` is preferred. The `max-age` is also passed-through to any downstream clients including CDNs and browsers.

- You can configure `minimumCacheTTL` to increase the cache duration when the upstream image does not include `Cache-Control` header or the value is very low.
- You can configure `deviceSizes` and `imageSizes` to reduce the total number of possible generated images.
- You can configure formats to disable multiple formats in favor of a single image format.

`minimumCacheTTL`

You can configure the Time to Live (TTL) in seconds for cached optimized images. In many cases, it's better to use a Static Image Import which will automatically hash the file contents and cache the image forever with a `Cache-Control` header of `immutable`.

next.config.js

```js
module.exports = {
  images: {
    minimumCacheTTL: 60,
  },
}
```

The expiration (or rather Max Age) of the optimized image is defined by either the `minimumCacheTTL` or the upstream image `Cache-Control` header, whichever is larger.

If you need to change the caching behavior per image, you can configure `headers` to set the `Cache-Control` header on the upstream image (e.g. `/some-asset.jpg`, not `/_next/image` itself).

There is no mechanism to invalidate the cache at this time, so its best to keep `minimumCacheTTL` low. Otherwise you may need to manually change the `src` prop or delete `<distDir>/cache/images`.

`disableStaticImages`

The default behavior allows you to import static files such as `import icon from './icon.png'` and then pass that to the `src` property.

In some cases, you may wish to disable this feature if it conflicts with other plugins that expect the import to behave differently.

You can disable static image imports inside your `next.config.js`:

next.config.js

```js
module.exports = {
  images: {
    disableStaticImages: true,
  },
}
```

`dangerouslyAllowSVG`

The default loader does not optimize SVG images for a few reasons. First, SVG is a vector format meaning it can be resized losslessly. Second, SVG has many of the same features as HTML/CSS, which can lead to vulnerabilities without proper Content Security Policy (CSP) headers.

Therefore, we recommended using the `unoptimized` prop when the `src` prop is known to be SVG. This happens automatically when `src` ends with `".svg"`.

However, if you need to serve SVG images with the default Image Optimization API, you can set `dangerouslyAllowSVG` inside your `next.config.js`:

next.config.js

```js
module.exports = {
  images: {
    dangerouslyAllowSVG: true,
    contentDispositionType: 'attachment',
    contentSecurityPolicy: "default-src 'self'; script-src 'none'; sandbox;",
  },
}
```

In addition, it is strongly recommended to also set `contentDispositionType` to force the browser to download the image, as well as `contentSecurityPolicy` to prevent scripts embedded in the image from executing.

`contentDispositionType`

The default loader sets the `Content-Disposition` ↗ header to `attachment` for added protection since the API can serve arbitrary remote images.

The default value is `attachment` which forces the browser to download the image when visiting directly. This is particularly important when `dangerouslyAllowSVG` is true.

You can optionally configure `inline` to allow the browser to render the image when visiting directly, without downloading it.

next.config.js

```js
module.exports = {
  images: {
    contentDispositionType: 'inline',
  },
}
```

Animated Images

The default loader will automatically bypass Image Optimization for animated images and serve the image as-is.

Auto-detection for animated files is best-effort and supports GIF, APNG, and WebP. If you want to explicitly bypass Image Optimization for a given animated image, use the unoptimized prop.

Responsive Images

The default generated `srcset` contains `1x` and `2x` images in order to support different device pixel ratios. However, you may wish to render a responsive image that stretches with the viewport. In that case, you'll need to set `sizes` as well as `style` (or `className`).

You can render a responsive image using one of the following methods below.

Responsive image using a static import

If the source image is not dynamic, you can statically import to create a responsive image:

components/author.js

```
import Image from 'next/image'
import me from '../photos/me.jpg'

export default function Author() {
  return (
    <Image
      src={me}
      alt="Picture of the author"
      sizes="100vw"
      style={{
        width: '100%',
        height: 'auto',
      }}
    />
  )
}
```

Try it out:

- Demo the image responsive to viewport↗

Responsive image with aspect ratio

If the source image is a dynamic or a remote url, you will also need to provide `width` and `height` to set the correct aspect ratio of the responsive image:

components/page.js

```js
import Image from 'next/image'

export default function Page({ photoUrl }) {
  return (
    <Image
      src={photoUrl}
      alt="Picture of the author"
      sizes="100vw"
      style={{
        width: '100%',
        height: 'auto',
      }}
      width={500}
      height={300}
    />
  )
}
```

Try it out:

- Demo the image responsive to viewport↗

Responsive image with `fill`

If you don't know the aspect ratio, you will need to set the `fill` prop and set `position: relative` on the parent. Optionally, you can set `object-fit` style depending on the desired stretch vs crop behavior:

app/page.js

```jsx
import Image from 'next/image'

export default function Page({ photoUrl }) {
  return (
    <div style={{ position: 'relative', width: '300px', height: '500px' }}>
      <Image
        src={photoUrl}
        alt="Picture of the author"
        sizes="300px"
        fill
        style={{
          objectFit: 'contain',
        }}
      />
    </div>
  )
}
```

Try it out:

- Demo the `fill` prop ↗

Theme Detection CSS

If you want to display a different image for light and dark mode, you can create a new component that wraps two `<Image>` components and reveals the correct one based on a CSS media query.

components/theme-image.module.css

```css
.imgDark {
  display: none;
}

```

```css
@media (prefers-color-scheme: dark) {
  .imgLight {
    display: none;
  }
  .imgDark {
    display: unset;
  }
}
```

components/theme-image.tsx TypeScript

```tsx
import styles from './theme-image.module.css'
import Image, { ImageProps } from 'next/image'

type Props = Omit<ImageProps, 'src' | 'priority' | 'loading'> & {
  srcLight: string
  srcDark: string
}

const ThemeImage = (props: Props) => {
  const { srcLight, srcDark, ...rest } = props

  return (
    <>
      <Image {...rest} src={srcLight} className={styles.imgLight} />
      <Image {...rest} src={srcDark} className={styles.imgDark} />
    </>
  )
}
```

Good to know: The default behavior of `loading="lazy"` ensures that only the correct image is loaded. You cannot use `priority` or `loading="eager"` because that would cause both images to load. Instead, you can use `fetchPriority="high"` ↗.

Try it out:

- Demo light/dark mode theme detection ↗

getImageProps

For more advanced use cases, you can call `getImageProps()` to get the props that would be passed to the underlying `<img>` element, and instead pass to them to another component, style, canvas, etc.

This also avoid calling React `useState()` so it can lead to better performance, but it cannot be used with the `placeholder` prop because the placeholder will never be removed.

Theme Detection Picture

If you want to display a different image for light and dark mode, you can use the `<picture>`↗ element to display a different image based on the user's preferred color scheme↗.

app/page.js

```
import { getImageProps } from 'next/image'

export default function Page() {
  const common = { alt: 'Theme Example', width: 800, height: 400 }
  const {
    props: { srcSet: dark },
  } = getImageProps({ ...common, src: '/dark.png' })
  const {
    props: { srcSet: light, ...rest },
  } = getImageProps({ ...common, src: '/light.png' })

  return (
    <picture>
      <source media="(prefers-color-scheme: dark)" srcSet={dark} />
      <source media="(prefers-color-scheme: light)" srcSet={light} />
      <img {...rest} />
    </picture>
  )
}
```

Art Direction

If you want to display a different image for mobile and desktop, sometimes called Art Direction ↗, you can provide different `src`, `width`, `height`, and `quality` props to `getImageProps()`.

app/page.js

```
import { getImageProps } from 'next/image'

export default function Home() {
  const common = { alt: 'Art Direction Example', sizes: '100vw' }
  const {
    props: { srcSet: desktop },
  } = getImageProps({
    ...common,
    width: 1440,
    height: 875,
    quality: 80,
    src: '/desktop.jpg',
  })
  const {
    props: { srcSet: mobile, ...rest },
  } = getImageProps({
    ...common,
    width: 750,
    height: 1334,
    quality: 70,
    src: '/mobile.jpg',
  })

  return (
    <picture>
      <source media="(min-width: 1000px)" srcSet={desktop} />
      <source media="(min-width: 500px)" srcSet={mobile} />
      <img {...rest} style={{ width: '100%', height: 'auto' }} />
    </picture>
  )
}
```

Background CSS

You can even convert the `srcSet` string to the `image-set()` ↗ CSS function to optimize a background image.

app/page.js

```
import { getImageProps } from 'next/image'

function getBackgroundImage(srcSet = '') {
  const imageSet = srcSet
```

```
      .split(', ')
      .map((str) => {
        const [url, dpi] = str.split(' ')
        return `url("${url}") ${dpi}`
      })
      .join(', ')
    return `image-set(${imageSet})`
  }

  export default function Home() {
    const {
      props: { srcSet },
    } = getImageProps({ alt: '', width: 128, height: 128, src: '/img.png' })
    const backgroundImage = getBackgroundImage(srcSet)
    const style = { height: '100vh', width: '100vw', backgroundImage }

    return (
      <main style={style}>
        <h1>Hello World</h1>
      </main>
    )
  }
```

Known Browser Bugs

This `next/image` component uses browser native lazy loading ↗, which may fallback to eager loading for older browsers before Safari 15.4. When using the blur-up placeholder, older browsers before Safari 12 will fallback to empty placeholder. When using styles with `width`/ `height` of `auto`, it is possible to cause Layout Shift ↗ on older browsers before Safari 15 that don't preserve the aspect ratio ↗. For more details, see this MDN video ↗.

- Safari 15 - 16.3 ↗ display a gray border while loading. Safari 16.4 fixed this issue ↗. Possible solutions:
  - Use CSS `@supports (font: -apple-system-body) and (-webkit-appearance: none) { img[loading="lazy"] { clip-path: inset(0.6px) } }`
  - Use `priority` if the image is above the fold
- Firefox 67+ ↗ displays a white background while loading. Possible solutions:

- Enable AVIF `formats`
- Use `placeholder`

Version History

Version	Changes
`v15.0.0`	`contentDispositionType` configuration default changed to `attachment`.
`v14.2.0`	`overrideSrc` prop added.
`v14.1.0`	`getImageProps()` is stable.
`v14.0.0`	`onLoadingComplete` prop and `domains` config deprecated.
`v13.4.14`	`placeholder` prop support for `data:/image...`
`v13.2.0`	`contentDispositionType` configuration added.
`v13.0.6`	`ref` prop added.
`v13.0.0`	The `next/image` import was renamed to `next/legacy/image`. The `next/future/image` import was renamed to `next/image`. A codemod is available to safely and automatically rename your imports. `<span>` wrapper removed. `layout`, `objectFit`, `objectPosition`, `lazyBoundary`, `lazyRoot` props removed. `alt` is required. `onLoadingComplete` receives reference to `img` element. Built-in loader config removed.
`v12.3.0`	`remotePatterns` and `unoptimized` configuration is stable.
`v12.2.0`	Experimental `remotePatterns` and experimental `unoptimized` configuration added. `layout="raw"` removed.
`v12.1.1`	`style` prop added. Experimental support for `layout="raw"` added.
`v12.1.0`	`dangerouslyAllowSVG` and `contentSecurityPolicy` configuration added.
`v12.0.9`	`lazyRoot` prop added.
`v12.0.0`	`formats` configuration added. AVIF support added.

Version	Changes
	Wrapper `<div>` changed to `<span>`.
`v11.1.0`	`onLoadingComplete` and `lazyBoundary` props added.
`v11.0.0`	`src` prop support for static import. `placeholder` prop added. `blurDataURL` prop added.
`v10.0.5`	`loader` prop added.
`v10.0.1`	`layout` prop added.
`v10.0.0`	`next/image` introduced.

Previous
<Form>

Next
<Link>

Was this helpful?

Stay updated on new releases and features, guides, and case studies.

you@domain.com Subscribe