Gatsby produces static content that can be hosted *anywhere* at scale in a cost-effective manner. This static content is comprised of HTML files, JavaScript, CSS, images, and more that power your great Gatsby application.

In some circumstances you may want to deploy *assets* (non-HTML resources such as JavaScript, CSS, etc.) to a separate domain. Typically this is when you're required to use a dedicated CDN for assets or need to follow company-specific hosting policies.

This `assetPrefix` functionality is available starting in gatsby@2.4.0, so that you can seamlessly use Gatsby with assets hosted from a separate domain. To use this functionality, ensure that your version of `gatsby` specified in `package.json` is at least `2.4.0`.

Usage

Adding to `gatsby-config.js`

```
module.exports = {
  assetPrefix: `https://cdn.example.com`,
}
```

One more step - when you build out this application, you need to add a flag so that Gatsby picks up this option.

Enable prefixing for builds

You must explicitly enable prefixing for a build by either adding the `--prefix-paths` flag or setting the `PREFIX_PATHS` environment variable. If this flag or env variable is not specified, the build will ignore this option, and build out content as if it was hosted on the same domain. To ensure you build out successfully, do one of the following:

```
gatsby build --prefix-paths
```

```
PREFIX_PATHS=true gatsby build
```

That's it! You have an application that is ready to have its assets deployed from a CDN and its core files (e.g. HTML files) can be hosted on a separate domain.

Building / Deploying

Once your application is built out, all assets will be automatically prefixed by this asset prefix. For example, if you have a JavaScript file `app-common-1234.js`, the script tag will look something like:

```
<script src="https://cdn.example.com/app-common-1234.js"></script>
```

However - if you were to deploy your application as-is, those assets would not be available! You can do this in a few ways, but the general approach will be to deploy the contents of the `public` folder to *both* your core domain, and the CDN/asset prefix location.

Using `onPostBuild`

You expose an `onPostBuild` API hook. This can be used to deploy your content to the CDN, like so:

```
const assetsDirectory = `public`

exports.onPostBuild = async function onPostBuild() {
  // do something with public
  // e.g. upload to S3
}
```

Using `package.json` scripts

Additionally, you can use an npm script, which will let you use some command line interfaces/executables to perform some action, in this case, deploying your assets directory!

In this example, the `aws-cli` and `s3` is used to sync the `public` folder (containing all the assets) to the `s3` bucket.

```
{
  "scripts": {
    "build": "gatsby build --prefix-paths",
    "postbuild": "aws s3 sync public s3://mybucket"
  }
}
```

Now whenever the `build` script is invoked, e.g. `npm run build`, the `postbuild` script will be invoked *after* the build completes, therefore making your assets available on a *separate* domain after you have finished building out your application with prefixed assets.

Additional Considerations

Usage with `pathPrefix`

The `pathPrefix` feature can be thought of as semi-related to this feature. That feature allows *all* your website content to be prefixed with some constant prefix, for example you may want your blog to be hosted from `/blog` rather than the project root.

This feature works seamlessly with `assetPrefix`. Build out your application with the `--prefix-paths` flag and you'll be well on your way to hosting an application with its assets hosted on a CDN, and its core functionality available behind a path prefix.

Usage with `gatsby-plugin-offline`

When using a custom asset prefix with `gatsby-plugin-offline`, your assets can still be cached offline. However, to ensure the plugin works correctly, there are a few things you need to do.

1. Your asset server needs to have the `Access-Control-Allow-Origin` header set either to `*` or your site's origin.
2. Certain essential resources need to be available on your content server (i.e. the one used to serve pages). This includes `sw.js`, as well as resources to precache: the webpack bundle, the app bundle, the manifest (and any icons referenced), and the resources for the offline plugin app shell.

You can find most of these by looking for the `self.__precacheManifest` variable in your generated `sw.js`. Remember to also include `sw.js` itself, and any icons referenced in your `manifest.webmanifest` if you have one. To check your service worker is functioning as expected, look in Application → Service Workers in your browser dev tools, and check for any failed resources in the Console/Network tabs.