How Shadowing Works

Shadowing is a powerful feature that allows theme users to override components, objects, and anything else in a theme's `src` directory.

> **Note** This is a technical deep dive into how Shadowing works. If you'd like to learn about what Shadowing is, see the What is Component Shadowing? blog post.

Shadowing works by using a webpack resolver plugin that maps themes in a `gatsby-config.js` to possible shadowed files. This gets especially mind melty because themes can add parent themes to a configuration so you need to be able to walk the composition of themes to determine the "last shadow" since the last shadowed theme file wins in the algorithm.

Theme Composition

It's important to begin discussing how the composition of themes works. An end user of a theme can configure any number of themes. Each of these themes are considered sibling themes. Here is a `gatsby-config.js` that configures two sibling themes:

```
module.exports = {
  plugins: ["gatsby-theme-tomato-blog", "gatsby-theme-tomato-portfolio"],
}
```

Both of the themes above (blog and portfolio) can install and configure any other theme so you end up with a tree of themes which we call a theme composition.

The theme composition itself has a few properties:

- the last theme wins
- a theme that uses another theme is the child theme
- a theme that is used by another theme is the parent theme
- theme trees are flattened during resolution

These characteristics are used in the component shadowing algorithm to decide which component to render. So, for example, if `gatsby-theme-tomato-blog` has `gatsby-theme-parent` as a parent theme it results in the following themes array:

```
const themesArray = [
  "gatsby-theme-parent",
  "gatsby-theme-tomato-blog",
  "gatsby-theme-tomato-portfolio",
]
```

This means that `gatsby-theme-tomato-portfolio` receives priority for component resolution, because it is last in the array.

Modifying the webpack Config

Component shadowing is a bit meta because it is implemented as an internal Gatsby plugin that applies a webpack plugin which modifies how module resolution happens for files that are shadowed.

The plugin consists of a `gatsby-node.js` and the webpack plugin code. The `gatsby-node` file is pretty straightforward:

```
const GatsbyThemeComponentShadowingResolverPlugin = require(`.`)

exports.onCreateWebpackConfig = (
  { store, stage, getConfig, rules, loaders, actions },
  pluginOptions
) => {
  const { flattenedPlugins, themes } = store.getState()

  actions.setWebpackConfig({
    resolve: {
      plugins: [
        new GatsbyThemeComponentShadowingResolverPlugin({
          extensions: program.extensions,
          themes: flattenedPlugins.map(plugin => {
            return {
              themeDir: plugin.pluginFilepath,
              themeName: plugin.name,
            }
          }),
          projectRoot: program.directory,
        }),
      ],
    },
  })
}
```

Structure of a webpack Plugin

The webpack plugin itself has a constructor and an apply function which webpack calls as part of module resolution. We tap into `resolve` hook and right before the "resolved" hook in the pipeline.

```
module.exports = class GatsbyThemeComponentShadowingResolverPlugin {
  constructor({ projectRoot, themes, extensions, extensionsCategory }) {
    this.themes = themes
    this.projectRoot = projectRoot

    // See more on extensions handling below
    this.extensions = ...
    this.extensionsCategory = ...
    this.additionalShadowExtensions = ...
  }

  apply(resolver) {
    // This hook is executed very early and captures the original file name
    resolver
      .getHook(`resolve`)
      .tapAsync(
        `GatsbyThemeComponentShadowingResolverPlugin`,
        (request, stack, callback) => {
          if (!request._gatsbyThemeShadowingOriginalRequestPath) {
            request._gatsbyThemeShadowingOriginalRequestPath = request.request
          }
          return callback()
        }
      )

    // This is where the magic really happens
    resolver
      .getHook(`before-resolved`)
      .tapAsync(
        `GatsbyThemeComponentShadowingResolverPlugin`,
        (request, stack, callback) => {
          // highlight-line
          // ...
        }
      )
  }
}
```

Identify requested theme and component

The `request` contains the path of the file that was requested. The first step is to extract from that path the theme to which that file belongs, as well as the local path of that file inside its theme's `src` directory.

```
resolver
  .getHook(`before-resolved`)
  .tapAsync(
    `GatsbyThemeComponentShadowingResolverPlugin`,
    (request, stack, callback) => {
      const [theme, component] = this.getThemeAndComponent(request.path)
    }
  )

getThemeAndComponent(filepath) {
  // find out which theme's src/components dir we're requiring from
  const allMatchingThemes = this.themes.filter(({ themeDir }) =>
    filepath.startsWith(path.join(themeDir, `src`))
  )

  // The same theme can be included twice in the themes list causing multiple
  // matches. This case should only be counted as a single match for that theme.
  const matchingThemes = _.uniqBy(allMatchingThemes, `themeName`)

  // 0 matching themes happens a lot for paths we don't want to handle
  // > 1 matching theme means we have a path like
  //   `gatsby-theme-blog/src/components/gatsby-theme-something/src/components`
  if (matchingThemes.length > 1) {
    throw new Error(
      `Gatsby can't differentiate between themes ${matchingThemes
        .map(theme => theme.themeName)
        .join(` and `)} for path ${filepath}`
    )
  }

  if (matchingThemes.length === 0) {
    return [null, null]
  }

  const theme = matchingThemes[0]

  // get the location of the component relative to its theme's src/
  const [, component] = filepath.split(path.join(theme.themeDir, `src`))

  return [theme, component]
```

```
}
```

The `resolved` hook is called after Webpack's default resolution process has been completed. At that point, the requested path has been resolved to the absolute path on disk of the file that would have been used if no shadowing was being performed. At that point, `node_modules`, aliases and symlinks have all been resolved. Also, the requested path will contain the file extension that was determined by Webpack.

For example, let's assume that user code requires a file named `gatsby-theme-tomato/src/button/heading`. On entering `before-resolved` hook for `GatsbyThemeComponentShadowingResolverPlugin`, `request.path` might looks something like `/some/path/my-site/node_modules/gatsby-theme-tomato/src/bu` (that is if `gatsby-theme-tomato` has been installed from a npm repository; that would be `/some/path/my-site/packages/gatsby-theme-tomato/src/button/heading.js` if yarn-style workspaces are being used). Now, assuming that theme `gatsby-theme-tomato` has been properly registered in that site's gatsby-config.js, then `getThemeAndComponent` will return:

```
;[
  {
    themeName: "gatsby-theme-tomato",
    themeDir: "/some/path/my-site/node_modules/gatsby-theme-tomato",
  },
  "button/heading.js",
]
```

Extracting the theme and local component's path so that we can determine the theme that is being required from so we can check for shadowed files in the user's site or other themes to match against. We also make sure the matched themes are unique because two themes can bring in the same theme to the theme composition. When that's the case we won't worry about them being different. Though, it is important to note that when performing resolution to build the site, the last theme added will always win.

Handle too many matches If there is more than one matching theme there is some sort of ambiguity and we should return an error. This can happen if there's a path like `gatsby-theme-blog/src/components/gatsby-theme-something/src/components` in the project.

No matches If there are no theme matches we return the invoked callback because there's nothing more to do, time to let webpack continue on its way with module resolution.

```
if (matchingThemes.length === 0) {
  return callback()
}
```

The component shadow

Once it is determined that a file being required belongs to a theme, we need to figure out if that file is actually shadowed by some other file (and if so, which path should it resolve to instead). We do this by calling `resolveComponentPath` which uses the theming algorithm to attempt to find a shadowed component. If nothing is found we let Webpack continue with its default resolution algorithm.

```
// This is the shadowing algorithm.
const builtComponentPath = this.resolveComponentPath({
  theme,
  component,
  originalRequestComponent,
})

if (builtComponentPath) {
  return resolver.doResolve(
    resolver.hooks.describedRelative,
    { ...request, path: builtComponentPath },
    null,
    {},
    callback
  )
} else {
  return callback()
}
```

If a shadowed component has been found, then we call `doResolve` on the resolver, telling it to jump back at the `describedRelative` hook. This is what tells webpack to resolve and bundle that particular file.

If no shadow component has been found, then we call `callback` instead, which tells Webpack to proceed to the next step in the current pipeline, without any change.

Resolving a shadowed component When looking for a component we perform a search that occurs in two locations:

- user's project
- themes

User's project In order to ensure that the user's project always takes precedence in terms of shadowing it's prepended to the theme list when attempting to resolve the component. This ensures that `my-site/src/gatsby-theme-tomato/box.js` will take priority over any other theme that might want to shadow the same component.

Themes As discussed before, themes are flattened into a list and then all possible shadow paths are constructed to match against. When concatenating with the user's project it's important to note again that the themes array is reversed. This is how we ensure that "the last theme wins" when resolving a shadowed file. We walk the list for matches from start to finish.

Additionally, the original theme is removed because that's the default behavior of webpack so we don't need to resolve a theme to itself.

```
const themes = this.themes.filter(
  ({ themeName }) => themeName !== theme.themeName
)

const themesArray = [
  path.join(this.projectRoot, `src`, theme.themeName),
].concat(
  themes
    .reverse()
    .map(({ themeDir }) => path.join(themeDir, `src`, theme.themeName))
)
```

Shadow Extensions Shadowing algorithm allows to use different extension than a file that is being shadowed. This does rely on theme using import paths that don't contain extensions (`./heading` imports would allow to use different extension while `./heading.jsx` would require to use exact same extension) and does rely on extension list that webpack can automatically resolve (via `resolve.extensions` webpack config option).

To get access to original request we tap into `resolve` hook and store `request` in `_gatsbyThemeShadowingOriginalRequestPath` field to have access to it later in `before-resolved` hook:

```
  apply(resolver) {
    // This hook is executed very early and captures the original file name
    resolver
      .getHook(`resolve`)
      .tapAsync(
        `GatsbyThemeComponentShadowingResolverPlugin`,
        (request, stack, callback) => {
          if (!request._gatsbyThemeShadowingOriginalRequestPath) {
            request._gatsbyThemeShadowingOriginalRequestPath = request.request
          }
          return callback()
        }
      )
```

Note however that `heading.css` would not logically be an acceptable shadow for `heading.js` since they are not providing the same type of content. Because

of this we have extension compatibility table:

```
const DEFAULT_FILE_EXTENSIONS_CATEGORIES = {
  // Code formats
  js: `code`,
  jsx: `code`,
  ts: `code`,
  tsx: `code`,
  cjs: `code`,
  mjs: `code`,
  coffee: `code`,

  // JSON-like data formats
  json: `json`,
  yaml: `json`,
  yml: `json`,

  // Stylesheets formats
  css: `stylesheet`,
  sass: `stylesheet`,
  scss: `stylesheet`,
  less: `stylesheet`,
  "css.js": `stylesheet`,

  // Images formats
  jpeg: `image`,
  jpg: `image`,
  jfif: `image`,
  png: `image`,
  tiff: `image`,
  webp: `image`,
  avif: `image`,
  gif: `image`,

  // Fonts
  woff: `font`,
  woff2: `font`,
}
```

We currently lack proper public API to extend this table, but it can be extended by passing the `extensionsCategory` property to the `GatsbyThemeComponentShadowingResolverPlugin` plugin in the Webpack configuration file. This requires that a Gatsby plugin (or the site itself) intercept the `onCreateWebpackConfig` event, then grab the existing Webpack configuration, find the existing `GatsbyThemeComponentShadowingResolverPlugin` entry in `resolve.plugins`, add or modify the `extensionsCategory` property, and finally call `setWebpackConfig` with the modified configuration. Note that

it should be considered internal API and it can break at any time so use this method with caution.

Finally, because the table is not exhaustive, we check extensions for the same category as shadowed file first before falling back to rest of extensions defined by `resolve.extensions` to not introduce breaking changes.

All together The shadowing algorithm can be boiled down the following function that's roughly 40 lines of code:

```
resolveComponentPath({ theme, component, originalRequestComponent }) {
  // don't include matching theme in possible shadowing paths
  const themes = this.themes.filter(
    ({ themeName }) => themeName !== theme.themeName
  )

  const themesArray = [
    path.join(this.projectRoot, `src`, theme.themeName),
  ].concat(
    themes
      .reverse()
      .map(({ themeDir }) => path.join(themeDir, `src`, theme.themeName))
  )

  const acceptableShadowFileNames = this.getAcceptableShadowFileNames(
    path.basename(component),
    originalRequestComponent
  )

  for (const theme of themesArray) {
    const possibleComponentPath = path.join(theme, component)
    debug(`possibleComponentPath`, possibleComponentPath)

    let dir
    try {
      // we use fs/path instead of require.resolve to work with
      // TypeScript and alternate syntaxes
      dir = fs.readdirSync(path.dirname(possibleComponentPath))
    } catch (e) {
      continue
    }
    const existsDir = dir.map(filepath => path.basename(filepath))

    // if no exact path, search for extension
    const matchingShadowFile = acceptableShadowFileNames.find(shadowFile =>
      existsDir.includes(shadowFile)
```

```
    )
    if (matchingShadowFile) {
      return path.join(
        path.dirname(possibleComponentPath),
        matchingShadowFile
      )
    }
  }
  return null
}
```

Handling component extending

This is where things begin to get a bit whacky. In addition to overriding a file, we want it to be possible to import the very component you're shadowing so that you can wrap it or even add props.

```
import React from "react"
import { Author } from "gatsby-theme-blog/src/components/author"
import Card from "../components/card"

export default function MyAuthor(props) {
  return (
    <Card>
      <Author {...props} />
    </Card>
  )
}
```

Learn more about extending components

This is the first case we'll handle when attempting to resolve the file.

In order to do this we need to leverage the **issuer** of the request. This points to the file that the request came from. This means it refers to *where* the `import` occurs. The **request** refers to what the import points to.

This is implemented by another method on the plugin's class which we call `requestPathIsIssuerShadowPath`. It checks all possible directories for shadowing of the requested component, and then returns whether the issuer's path is one of them. Let's first take a look at the code and then unpack what's happening here.

```
requestPathIsIssuerShadowPath({ theme, component, issuerPath, userSiteDir }) {
  if (!theme || !component) {
    return false
  }

  // get list of potential shadow locations
```

```
  const shadowFiles = this.getBaseShadowDirsForThemes(theme.themeName)
    .concat(path.join(userSiteDir, `src`, theme.themeName))
    .map(dir => path.join(dir, component))
    .flatMap(comp => this.getAcceptableShadowFileNames(comp))

  // if the issuer is requesting a path that is a potential shadow path of itself
  return shadowFiles.includes(issuerPath)
}
```

In the above code block `getBaseShadowDirsForThemes` returns:

```
const baseDirs = [
  "/Users/johno/c/gatsby-theme-example-component-extending/gatsby-theme-rebeccapurple/src/ga
  "/Users/johno/c/gatsby-theme-example-component-extending/gatsby-theme-tomato/src",
]
```

This constructs the shadowable files for `gatsby-theme-tomato`'s Box component. Then, we join the component path and end up with:

```
const fullPaths = [
  "/Users/johno/c/gatsby-theme-example-component-extending/gatsby-theme-rebeccapurple/src/ga
  "/Users/johno/c/gatsby-theme-example-component-extending/gatsby-theme-tomato/src/box",
]
```

We then know that if the issuer *matches* one of these components, then issuer itself is a shadow of the file being required. When this happens, we return the next path, so here the original location of the theme: `/Users/johno/c/gatsby-theme-example-component-extending/gatsby-theme-tomato/src/box`.

This means that when our shadowed file imports Box from a shadowed file we return the original box component defined in the theme.

As a result, the following will work as we expect:

```
import React from "react"
import Box from "gatsby-theme-tomato/src/box"
import Card from "../components/card"

export default function MyBox(props) {
  return (
    <div style={{ padding: "20px", backgroundColor: "rebeccapurple" }}>
      <Box {...props} />
    </div>
  )
}
```

Now, all usages of the Box in `gatsby-theme-tomato` will be also wrapped in a purple box.

An edge case If a theme sets `module` config the issuer will be null. As such we need to first check that the `request.context.issuer` is present before we attempt to resolve the shadowed component.

It's important to note that we don't recommend appending to the modules list in themes. Though, if you do, we will make sure we don't arbitrarily error.

Summary

Shadowing uses a predictable algorithm that leverages webpack to dynamically change module resolution based on a `gatsby-config` and theme composition. The last theme will take precedence in the shadowing algorithm, and the user's `src` directory is always take into account first.