Maximize Theme Performance

When designing themes, it is important to keep page load times to a minimum. Small load times result in a range of benefits, chief among which are shopper conversions and better SEO results. Consider the following guidelines to minimize page load times for your theme (for both the customer experience and monitoring tools such as Lighthouse).

Use Your Browser's Developer Tools

As a general best practice for theme development, take advantage of the profiling and network analysis features available through your browser's developer tools. Chrome and Safari have very extensive performance analysis tools, for example, and Firefox isn't far behind. There are official tutorials available online, and taking just a little time to familiarize yourself with these tools provides you with the means to quickly gather summary data about your theme's network performance.

Ensure Debug Mode is Off on Production Sandboxes

Debug mode is useful during development, but prevents performance optimizations from occurring when enabled on a production environment. When you turn debug mode off, the following optimizations occur:

- The LESS preprocessor produces minified and compressed CSS that excludes filename comments and line numbers.
- Cache headers are set for images and other resources to optimize load times.
- Storefront errors display generic error messages rather than detailed debugging information.
- The `trailing-scripts.hypr` template in the `templates/modules` directory calls minified rather than debug versions of RequireJS, HyprLive, and the Storefront SDK, loading the requested modules from the `compiled/scripts` directory rather than loading separate uncompiled modules at runtime as is the case during debugging.

The recommended method to enable and disable debug mode is to use the `debugMode` URL parameter. This parameter toggles debug mode using a cookie on your browser. For example: `www.yourSite.com/gadgets?debugMode=false` turns debug mode off for your browsing session.

A deprecated theme setting which you can configure in `theme.json`, `themeSettings.useDebugScripts`, allows you to persist debug mode across your entire site. The disadvantage of this setting compared to the `debugMode` URL parameter is that debug mode can remain on in a production environment after your browsing session has expired if you forget to turn it off. For this reason, the use of `themeSettings.useDebugScripts` is not recommended. Instead, use the recommended `pageContext.isDebugMode` setting to check whether debug mode is enabled (through the recommended URL parameter method) and write conditional logic for cases when it is enabled.

To confirm that debug mode is off and that there are no issues caused by the deprecated `useDebugScripts` theme setting, complete the following procedure:

1. Use a browser to view a storefront site running your theme.

2. Ensure debug mode is off by appending `debugMode=false` as a URL parameter.
3. Using your browser's developer tools, view the network traffic to confirm that the `require-min.js` script is running.
4. If instead of the `require-min.js` script you see `require-debug.js`, search your theme templates to replace any instances of `themeSettings.useDebugScripts` with `pageContext.isDebugMode`. The Core theme has one exception to this rule in the trailing-scripts.hypr template, within the opening `with` tag. You do not need to change this instance.
5. Save any changes to your theme and repeat this procedure to confirm that the correct script is running and that your site is therefore benefiting from the performance optimizations that occur when debug mode is off.

Mark New Scripts for Minification

Unlike other theme optimizations, which the storefront generally performs at run time, script compilation occurs on the client side during the Grunt build process that lints and uploads theme files to Dev Center. The Grunt workflow uses a customized version of RequireJS that walks the script dependency tree created by `define()` calls in your theme's JavaScript modules.

The build process, however, does not automatically detect scripts loaded in templates using the `require_script` tag. Instead, the build process reads the `build.js` configuration file in your theme's root directory to identify which scripts to minify on your site.

If you add new scripts to your theme using the `require_script` tag, mark them for minification using the following procedure:

1. Use a browser to view a storefront site running your theme.
2. Open your browser's developer tools.
3. View the network monitor and filter the traffic for script files.
4. Navigate through your site so that you view all the available page types.
5. As you navigate around your site, use the network monitor to identify scripts generated by your theme that fit the following criteria:
    - Scripts generated by your theme templates have a `cacheKey` appended to them. If this key is not present, the scripts are not generated by a theme template, so you do not need to take note of them.
    - Exclude scripts initiated by `require-min.js`. The build process has already minified these scripts.
    - For any scripts that meet the preceding criteria, note whether the script occurs across multiple pages or in one page only.
6. Open the `build.js` file in your theme's root directory.
7. Add the scripts you identified as common to multiple pages to the `modules/common` group within the `modules` array. For example:

```
modules: [
  {
    name: "modules/common",
    include: [
      'modules/api',
      ...
      'modules/yourCoolCommonScript',
    ],
    exclude: ['jquery'],
  },
```

8. Add the scripts you identified as unique to one page to the corresponding group within the `modules` array. For example:

```
modules: [
  {
    name: "modules/cart",
    include: [
      'modules/yourCoolCartScript',
    ],
    exclude: ['modules/common'],
  },
```

9. After saving your changes, run `grunt` to build and upload your theme files.
10. In your browser, navigate around your storefront site again. Confirm that the extra scripts have been bundled into the minified files and that all functionality still works as expected.

Detect Slow Third-Party Code

Use your browser's developer tools to determine whether poor site performance is due to third-party code.

Third-party code is common on e-commerce sites, delivering functionality in the form of affiliate networks, analytics frameworks, social networking integrations, chat apps, review platforms, and other useful services. For a variety of reasons, third-party code may at times suffer from poor performance.

To troubleshoot whether third-party code is causing adverse effects on your site:

1. Use a browser to view a storefront site running your theme.
2. Open your browser's developer tools to:
    1. View the network monitor and order the results by time to identify slower requests.
    2. View the timeline monitor to identify files or functions that cause delays during a full page load.
    3. View the CPU profiles monitor to collect a CPU snapshot and identify which files use the most CPU resources.
3. If third-party code appears to be responsible for a performance issue or delay, test the theme with the third-party code temporarily factored out and note the difference in performance.

If you isolate third-party code as the root cause of slow site performance, you should send your findings to the third-party provider. However, before you initiate contact, you should double-check the implementation documentation for the third-party code to make sure a configuration issue is not at fault. Also, if possible, you should examine the

performance of the third-party code on non-Kibo sites to determine whether the issue is inherent to the code or unique to the Kibo implementation.

Remove Unused Code and Plugins

In addition to checking for slow third-party code, you should also remove any code that isn't being used by the theme and site pages. This unnecessary code can cause your site to load more slowly or introduce unexpected errors.

Kibo does not recommend adding plugins to your site for similar reasons, as plugins can negatively affect performance by introducing slow third-party code, unnecessary code content, or integration errors.

Leverage Server-Side Rendering For Catalog Pages

The Hypr templating system renders all content on the server, but allows JavaScript code in HyprLive templates (designated by the `.hypr.live` extension) to leverage the Storefront SDK and communicate with the Kibo API to render new data on a page after initial server rendering. However, for catalog pages, where SEO is paramount, Kibo recommends that you minimize the use of JavaScript in creating the view.

For best results, review the HyprLive templates in your theme to determine whether you can achieve the same functionality using a Hypr tag or filter. For example, you can replace client-side scripts that call the Product Search API, Document Lists API, or Entities API with the include_products, include_documents, or include_entities tags, respectively.

Avoid Excessively Large Images

As a general rule of thumb, no image that is not a full-bleed background image should exceed 1 MB in size. If an image breaks this rule, make sure it is saved using the appropriate file type.

- Is it a line graphic or does it contain mostly text?

  Recommended File Types: PNG, GIF

- Is it a photograph or does it contain many colors or gradients?

  Recommended File Types: JPEG

- Does it combine photographic images with sharp logos or text?

  Recommended File Types: JPEG for the background, transparent PNG for the overlay

In addition the general rule on image size, note that the image manipulation fields available to you in do not work on images that contain dimensions greater than 30,000 pixels in length.

An appealing e-commerce site has large, beautiful imagery, and therefore consumes most of its bandwidth by serving images. As long as pages load quickly and the majority of your images are below 1 MB in size, don't compress images to the detriment of display quality.

Use CSS Instead of JavaScript DOM Manipulation

CSS rule evaluation is one hundred times faster than JavaScript DOM manipulation. Although complex CSS rules may be hard to maintain, they almost always result in better performance compared to achieving the same styling through JavaScript.

Use Metrics to Evaluate Slow Code Paths

You don't know how fast your JavaScript is until you profile it. Identify slow code paths with performance analysis tools and fix code only if the numbers indicate a performance hit.

JavaScript engines in modern browsers optimize code aggressively. A JavaScript pattern that you've learned to avoid because of poor performance in older browsers can be very fast because of these optimizations. Do not rely on intuition or old advice to touch up your JavaScript.

Put JavaScript at the Bottom of the Page

Kibo also recommends putting JavaScript calls near the bottom of the page to reduce load time and improve responsiveness, so that the rest of the page content is displayed before the JavaScript is processed.

Remember that Sandboxes are Slower by Design

Your theme may perform more slowly than you would expect when you preview it on a development sandbox. However, rest assured that your theme should perform faster in a production tenant. Keep in mind that sandboxes:

- Don't use the same fast hardware that production tenants do. Because sandboxes are free and unlimited, they share a limited set of computing resources.
- Have caching layers turned off.
- May be using a browsing session with debug mode enabled, which results in slower performance.

Test Against Core Theme

You should test your custom theme in comparison to Kibo's core theme before rolling it out on production, to make sure that all functionality is behaving as expected.

For instructions on how to test an identical version of your site on the core theme instead of your custom theme, see the Troubleshooting Your Implementation guide. That guide focuses on determining whether a bug is originating from Kibo's core version of UCP or an issue in your custom theme, but is also useful for general troubleshooting and testing purposes.

Inform Kibo of Expected Traffic

If you have planned promotional activities that may result in a significant increase to server traffic and/or API requests in a short period of time, Kibo recommends the following best practices:

- Do not send out a major email or text campaign to everyone in a large audience at the exact same time. Instead, space your emails and texts out over a period of time (1-2 hours).
- Send your emails and texts at a time that does not already have high traffic. For instance, most sites see a steady increase in traffic from 7am to 11am Central, so we would recommend that you do not send out an email campaign to several million customers at 8:30am.
- Please read and understand our recommended best practices for API related processes.

Doing the above will ensure the system can best scale on its own to meet your needs and avoid potential problems. In the event that a large spike or increase in traffic/requests cannot be avoided, please notify . Support will work with the developer teams to ensure Kibo has the resources in place to support the increase in traffic without being surprised by unexpected system load and negatively affecting performance.

Examples of these planned promotional activities include:

- National television or radio campaigns
- Publicity events such as sponsor interviews
- Spots on morning television
- Email marketing campaigns or widely advertised sales