JavaScript

JavaScript is an essential internet technology. When developing themes, you will likely use JavaScript in the following ways:

- On your storefront, in the shopper's browser (using HyprLive).
- In the editors that display custom menus in Admin.
- Through the Node.js theme tools that Kibo provides to facilitate development.
- Within API Extension applications, if you wish to explore deeper customization options and truly extend the platform.

This topic focuses on how JavaScript runs on the storefront and how to best take advantage of it when designing your theme.

How Your Storefront Leverages JavaScript

You are already very familiar with how JavaScript is used by most websites on the internet. When JavaScript runs on your storefront, it does so in the traditional sense that you've grown used to. JavaScript adds functionality and complex behavior to pages as they render and execute on the shopper's browser. To name just a few examples, JavaScript adds popover links during the login and signup process, fills in text when a shopper enters a query, runs Google Analytics to gather valuable data, provides filtering enhancements on category pages, enables image carousels, and renders complex pages like the checkout page, which contains more JavaScript than Hypr code.

In general, JavaScript enables you to:

Manipulate the Document Object Model (DOM)	Themes use jQuery, a well-understood, heavily-tested, and robust JavaScript library that allows you to traverse HTML documents, manipulate elements, handle events, inject animations, and utilize a variety of plug-ins.
Generate new view content for templates	In addition to Hypr templates, themes leverage HyprLive templates, denoted by the `hypr.live` extension. HyprLive is a JavaScript library that extends the Hypr templating language. After the initial server-side rendering of templates, HyprLive renders new content on a shopper's browser in response to specified changes and events brought upon as the shopper interacts with your site.
Get and set data through AJAX and XML/HTTP requests	The Storefront SDK helps you obtain complicated data from the API, which is designed for data-heavy business systems. The Storefront SDK uses AJAX and reverse proxies to make API calls, target the correct URL endpoints, and expose friendly methods, like `productAddToCart(quantity);`, to make development easier.

Asynchronous Module Definition

Despite the invaluable benefits of JavaScript, it does have disadvantages. Namely:

- An enterprise website loads a large number of scripts, which requires lots of HTTP requests and impacts the load time of your site.
- Scripts load sequentially depending on when they are called, which means that other threads or page requests are blocked from loading until the JavaScript is in place.
- If your site loads slowly, the shopper experience is less pleasing, visits are less likely to convert into sales, and search engines penalize you in regards to SEO.

To address these issues, themes use an asynchronous module definition (AMD). AMD treats scripts as modular units within a comprehensive list of dependencies, and loads required scripts asynchronously to prevent bottlenecking. With AMD, the JavaScript files that you require in your templates are loaded in the most efficient order possible, in as few scripts as possible, without multiple calls to the same script, and without a sequential loading constraint that prevents other tasks from executing while a script loads.

RequireJS

Specifically, themes leverage the RequireJS module loader. RequireJS minifies all the JavaScript files that you have marked as required in your templates and scripts (with few exceptions, such as scripts hosted by third parties), and loads the resulting modules asynchronously from the compile pile. RequireJS walks the array of dependencies, which means that if Script B requires Script A, you can just require Script B in your template and RequireJS determines that Script A is also needed.

Your theme loads RequireJS in the trailing-scripts.hypr template—which is included by the `page.hypr` base template—using a `<script>` tag, followed by basic configuration parameters inside the `require.config` object. With this code in place, you have access to two functions provided by RequireJS: `require` and `define`. You use these functions to specify dependencies in your script files. The `trailing-scripts.hypr` template also implements the `all_scripts` tag, which compiles a list of script dependencies in your templates.

List Script Dependencies in a JavaScript File

When you write a script file, you can use either `require` or `define` to list dependencies.

Function	When To Use It
`require`	Use this function in scripts that are not required by other scripts. For example,checkout.js is not required by another script and therefore us the beginning of the file to list dependencies: `require(['modules/jquery-mozu', 'underscore', 'hyprlive', 'modules/backbone-mozu', 'modules/models-checkout', 'modules/views-messages', 'modules/cart-monitor', 'hyprlivecontext', 'modules/editable-view', 'modules/preserve-element-through-render'], function ($, _, Hypr, Backbone, CheckoutModels, messageViewFactory, CartMonitor, HyprLiveContext, EditableVi { ...`
`define`	Use this function in scripts that are required by other scripts. For example,models-checkout.js is required by `checkout.js` (it returns colle `checkout.js` uses), and therefore uses the following syntax at the beginning of the file to list dependencies : `define(['modules/jquery-mozu', 'underscore', 'hyprlive', 'modules/backbone-mozu', 'modules/api', 'modules/models-customer', 'modules/models-address', 'modules/models-paymentmethods', 'hyprlivecontext'], function ($, _, Hypr, Backbone, api, CustomerModels, AddressModels, PaymentMethods, HyprLiveContext) { ...`

Themes include popular modules like jQuery and Backbone.js, and these scripts are also called using the RequireJS programming structure (for example, jquery-mozu.js and backbone-mozu.js).

List Script Dependencies in a Hypr Template

Use the `require_scripts` tag to list script dependencies in a Hypr template. During the build process, the `all_scripts` tag in the `trailing-scripts.hypr` template scans all your templates and assembles a list of the required scripts for RequireJS to load. The following is an example of using the `require_scripts` tag (which expects the script path to be within the `scripts` directory) in a template to specify a script dependency:

```
{% extends "page" %}
...
{% block body-content %}
   {% require_script "pages/checkout" %}
...
{% endblock body-content %}
```

If you use the `require_scripts` to require a tag that is not already a default script included by your theme's `build.js` file, you must add the script to that file.