



SWIFTOTTER'S JAVASCRIPT STUDY GUIDE

JESSE MAXWELL
SWIFT OTTER STUDIOS



SWIFTOTTER
SOLUTIONS

INTRODUCTION

You downloaded the most comprehensive resource for Magento 2 Javascript Development. The material covered provides excellent methodology for training developers who are new to these concepts.

This study guide answers the questions presented in Magento's study guide for [this test](#). Reading this study guide is part of solid preparation for passing the test but does not guarantee a passing grade. What will yield a passing grade is extensive experience with Magento's Javascript system combined with rigorous and careful review of this study guide.

Most people who spend \$260 on a test want confidence that they are likely to pass it. Therefore the next step of taking our [practice test](#) is critical. The practice test validates that you are ready for the test. Following completion of the practice test, you see questions you answered correctly and the ones you didn't.

Additionally, for a small fee, you can see your scores on the practice test by objective (showing your weak areas) and get the objective scores emailed to you. Think about it. After you take the real test, don't you want to know where you were weak? Now, you can know before you take the test how you are doing and where to study more.

Acknowledgements

Jesse, my brother, spent many hours writing this study guide. He researched topics that aren't inherently clear to make it easier for you. He has mastered the trickier aspects of Magento Javascript development and is investing back into you by sharing this knowledge. Please take a moment to thank him on [Twitter](#) or [email](#).

We also want to acknowledge Vinai Kopp's Mage2.tv as an excellent resource for learning Magento Javascript development. His in-depth videos are invaluable and have greatly beneficial. Please consider subscribing to Mage2.tv.

All the best!

Joseph Maxwell

WE ARE SWIFTOTTER

We are focused, efficient, solution-oriented and use Magento as the agent for solving medium-sized ecommerce merchant's challenges. New sites, migrations and maintenance are our bread and butter. Our clients are our friends. Simple.

In addition, we provide second-level support for merchants with in-house development teams. While moving development in-house can save money, it often leaves holes in the support system. We patch those holes by being available to quickly solve problems that might be encountered.

This study guide demonstrates our commitment to excellence and our love for continuous learning and improvement. Enhancing the Magento developer community is good for everyone: developers, agencies, site owners and customers.

DRIVER

[Driver](#)—transforming your production database to staging and local.

There are a number of tools to assist in getting production data back to staging. One of these is [Sonassi's](#). They work well.

But, there are some inherent shortcomings:

- These are limited by your skill level with `sed`.
- No way to run SQL commands.
- Transformations happen in the production environment.

What if you could:

- Run custom SQL commands to generate custom output for multiple environments?
- Automatically upload to S3?
- Store configuration in git, making it easy to replicate and adjust?

We have built a tool just for that: Driver.

Driver allows you to do all the above. For example, we generate data for three environments: staging, local initialization and local refresh. For the latter two environments, the customer and order data has been so sanitized that we have Driver set a standard admin login and password. The only thing that touches the production database is the initial `mysqldump` command.

This process has become so automated that we can run one command on our development machine to refresh the data: `reload_customer_name`.

How does it work?

The way Driver works is the production data is dumped and uploaded to a fresh Amazon RDS (relational data storage) instance. The transformations are run for each environment; that environment is dumped and loaded into S3. You can easily automate the synchronizations back to your staging and local environments.

<https://github.com/SwiftOtter/Driver>

CI

Are you familiar with Jenkins automation? Maybe you are having trouble getting code to production. We have developed an open-source continuous integration and deployment system built on Jenkins to make building and deploying Magento 1 and 2 websites easy.

Additionally, our system enables deploying Magento 2 code with zero downtime (or very little—if database updates are necessary).

Go to [our website](#) for more information.

CONTENTS

Introduction.....	2
We Are SWIFTotter	4
1 Technology Stack.....	8
1.1 Demonstrate understanding of RequireJS	9
1.2 Demonstrate understanding of UnderscoreJS	16
1.3 Demonstrate understanding of jQuery UI widgets	19
1.4 Demonstrate understanding of KnockoutJS	22
2 Magento JavaScript Basics.....	27
2.1 Demonstrate understanding of the modular structure of Magento	28
2.2 Describe how to use JavaScript modules in Magento.....	31
2.3 Demonstrate ability to execute JavaScript modules	37
2.4 Describe jQuery UI widgets in Magento	43
2.5 Demonstrate ability to customize JavaScript modules	47
3 Magento Core JavaScript Library.....	52
3.1 Demonstrate understanding of the mage library	53
3.2 Demonstrate understanding of mage widgets	61
3.3 Demonstrate the ability to use the customer-data module....	65
4 UI Components	69
4.1 Demonstrate understanding of Knockout customizations	70
4.2 Demonstrate understanding of Magento UI components.....	77
4.3 Demonstrate the ability to use UI components	86
4.4 Demonstrate understanding of grids and forms	90
5 Checkout	103
5.1 Demonstrate understanding of checkout architecture	104
5.2 Demonstrate understanding of payments	109
Exam Notes:.....	112
Glossary	113

1. TECHNOLOGY STACK



1.1 DEMONSTRATE UNDERSTANDING OF REQUIREJS

Describe the RequireJS framework and its approach to JavaScript module organization

RequireJS facilitates segmenting Javascript into asynchronously loaded modules. These modules are loaded and executed with no specific order except when it relates to dependencies. RequireJS provides a way to specific dependencies for modules. The dependencies are loaded before the module is executed so it can run without having to cautiously check for other code. All of RequireJS happens at runtime in the browser when the page is loaded (or some other action is initiated).

What is the main purpose of the RequireJS framework?

RequireJS allows deferred execution for the majority of the Javascript on the page by asynchronously requesting modules. Loading JavaScript via RequireJS allows a developer to ensure that required modules are accessible when referencing them. RequireJS also allows grouping Javascript code into logical modules, making writing and maintaining Javascript easier.

What are the pros and cons of the AMD approach to JavaScript file organization?



AUTHOR NOTE

These pros and cons are in comparison to the Magento 1 technique of including script tags directly into the head. They are not meant to compare AMD with newer tools such as Webpack.

Pros:

- Javascript can be loaded asynchronously so it is not render-blocking on page load.
- Logical groups of code are broken into modules so code is easier to understand.
- Smaller amounts of code can be overridden for the same effect because files are smaller.
- Decreases time to first paint (although it can make the page's Javascript-powered functionality unusable for more time).
- Ensures the required module is present when referencing it.
- JavaScript is only loaded when it's required, where traditionally a library such as jQuery may be included in the head and always loaded with RequireJS, it will only be loaded if it is required by reference on the page in question.

Cons:

- AMD is not a real standard, just a proposition that gained traction in the Javascript community.
- Code has to be wrapped in a boilerplate outer function.
- There is little control over the size of the modules because they are individually loaded on the frontend. It is more efficient to group them to some degree to minimize total requests. (Magento provides a way to group the modules for deployment, but this comes with side effects that more than negate any benefit.)

Which capabilities does RequireJS provide to create and customize JavaScript modules?

RequireJS allows you to easily build and inject Javascript modules into a site without incurring blocking page render. Modules can have dependencies without needing complex configuration or conditional loading logic. There is a powerful

configuration layer which allows you to change which dependencies are loaded, or add modules without needing to change the original Javascript files.

Demonstrate the ability to use `requirejs-config.js` files in the development process

What is a `requirejs-config.js` file?

It is a Javascript configuration file that is concatenated with all the other `requirejs-config.js` files for a given area (i.e. “frontend” or “adminhtml”). The merged file is generated on the server with PHP collecting all `requirejs-config.js` files for an area from modules and themes and combining them into a single file before being saved. The location of the file influences its order of execution; [see the docs for the specific order](#).

`requirejs-config.js` files follow the standard of:

```
var config = {  
    //...  
}
```

In which cases it is necessary to add configurations to it?

It is perhaps most helpful for modifying core code by overriding modules and for adding mixins (described in more detail later). Essentially, any time updating dependencies is necessary, `requirejs-config.js` is usually a good choice because it does not require changing the Javascript modules themselves.

What tools does it provide?

There are quite a few things you can do using only the configuration file but following are some of the ones more commonly used throughout Magento. We suggest you [refer to the RequireJS docs](#) for a comprehensive list.

- **map**: the object assigned to the **map** key allows substitution of module IDs. For instance, the ID of a commonly used module is `'Magento_Ui/js/lib/core/element/element'`. In the `requirejs-config.js` for that module, however, is an entry which allows other modules to use a much more succinct `uiElement` to require the module:

```
var config = {  
    map: {  
        '*': {  
            uiElement: 'Magento_Ui/js/lib/core/element/element'  
        }  
    }  
};
```

Note the `'*'` item in the **map** object. Inside the **map** object, keys target specific modules with the exception of the asterisk: it (*) is used to target *all* modules. When a module ID is used, however, it provides the powerful flexibility of being able to change dependencies in *a* or *any specific* module.

- **path**: this provides path mappings for module names not found directly under `baseUrl`. It is similar, but ultimately less targeted than the **map** key. In other words, the **path** object is more global and will change all references to a portion

of a path. Consider: `{"Old_Module/prototype": "New_Module/lib"}`. Now, a module reference of `"Old_Module/prototype/js/toolkit"` will actually load `"New_Module/lib/js/toolkit"`.

The following is from the `Magento_Ui` theme (random aside: this configuration item attempts to patch the lone and confusing “templates” directory used in contrast to the typical, singular “template” directory where HTML templates are placed [reference](#)):

```
var config = {  
  paths: {  
    'ui/template': 'Magento_Ui/templates'  
  }  
};
```

- `config`: used for passing configuration into modules and, importantly, in Magento, for declaring mixins (described in more detail later).
- `shim`: configures dependencies, exports, and custom initialization for older, traditional "browser globals" scripts that do not use `define()` to declare the dependencies. An example case for this configuration option would be to include a custom module as a dependency for a core module so it runs every time the core one does. Another good use case for this is when working with older libraries that don't support AMD.

In the following example, it is important that `jquery/jquery.cookie` load before `jquery/jquery-storageapi`. While this would often be handled in the module directly, we can also assert the load order here. Remember, RequireJS has no particular load order unless specifically provided.

```
shim: {  
  'jquery/jquery-storageapi': {  
    'deps': ['jquery/jquery.cookie']  
  }  
}
```

What are global callbacks?

This is a function that is executed after `deps` have loaded. As of the time of this writing, it isn't used in the core but still could be leveraged.

How can mappings be used?

More information is provided above. Using the `map` configuration object allows you to selectively replace modules or update their location. It also allows you to provide a shorter alias for a module.

Demonstrate the ability to use RequireJS plugins

What are RequireJS plugins?

Plugins provide additional functionality when loading modules. Plugins can be recognized in the core by an `!` (exclamation mark) preceding a module ID. Like so: `'text!./templates/virtual.html'`. The plugin name is `text` and handles [loading textual files](#). An HTML template is a good use case because it is loaded as plain text before being rendered as HTML. Ultimately, loader plugins are just another module, but they implement a specific API.

When a plugin is used, that plugin module is loaded first. The dependency's name is passed into a `load` method—the only one required in the plugin. The `load` method then handles processing the string it receives.

The [RequireJS docs are a good place](#) to find more details.

What are the `text` and `domReady` plugins used for?

The `text` loader plugin processes a module as text. This allows HTML to be put into separate templates which can be easier than managing unwieldy multi-line strings in a Javascript file.



Note that [template literals](#) provide a much easier way to include multi-line strings directly in a Javascript file if the situation warrants doing so. However, Internet Explorer 11 does not support them. They are helpful in more situations than only multi-line strings, though, and we will dive more into Magento's use of them later in this guide.

The `domReady` plugin waits for the DOM to load before executing the module. This is often not essential, but if your module has a dependency on the DOM (i.e. `document.querySelector('.class-name')`) it is good practice to include because it removes the possibility of strange race conditions.

1.2 DEMONSTRATE UNDERSTANDING OF UNDERSCOREJS

Demonstrate understanding of underscore utility functions

[UnderscoreJS](#) is a popular cross-browser library containing many miscellaneous helper functions including ones that assist with templating. It can be loaded into Javascript modules by adding a dependency for `'underscore'`. The parameter in the callback is usually simply `_` then. The functions are well documented on the UnderscoreJS website: <https://underscorejs.org/>.

What are the benefits of using the underscore library versus native JavaScript?

The goal of a library is to provide already-reliable functions for common tasks so you don't have to reinvent the wheel every time. It's worth keeping in mind that many of these functions are now available natively in Javascript.

One idea, then, is that UnderscoreJS is beneficial because it provides cross-browser support for these functions. However, due to the tremendous progress browsers have made, we find it difficult to say that using a library for helper functions is superior to using native Javascript. Consequently, we recommend using native Javascript *anytime it serves your intended audience*. In other words, if all the browsers you need to support offer a method that you need, it is better to use the native method because the library version adds processing overhead, with no other benefits. You might be surprised at how many APIs are now commonplace across the major browsers with Internet Explorer 11 being the main exception. Internet Explorer 11's use is dropping so low that many companies are dropping support for it on their websites. (If you really want to stretch this, [you may be able to go so far](#)

[as to say you are doing your users a favor to drop IE 11 support](#) and encourage them to upgrade to a more secure browser.)

Good examples of well supported APIs are `Array.map()` and `Array.filter()`—both of which are supported by IE 11 and all other major browsers. Others like `Array.find()` are supported in all major browsers except IE 11.

With that note aside, it is better to use the library than to write a custom function to do the same thing. The benefit of using UnderscoreJS, in that case, is that it is fewer lines of code you have to debug and maintain.

Use underscore templates in customizations



FURTHER RESOURCES:

Refer to [the docs](#) for the specific usage of templates.

There are three styles of handling input variables:

- `<%= ... %>` -- interpolates a value
- `<%- ... %>` -- interpolates a value and escapes HTML
- `<% ... %>` -- executes arbitrary Javascript

Describe how underscore templates are used.

UnderscoreJS's `.template()` method is central to Magento's template rendering and is used broadly through the core via the `lib/web/mage/template.js` (referenced as a dependency with `mage/template`). The `.template()` method is used by passing a string into it. A callback is returned and is executed with

an object for context. If you find a cool trick [in the docs](#) for this template, you're probably safe to use it.

For example: `testString = _.template('Test string with the following variable: <%- example %>').Then, testString({example: 'some text'})`.

What are the pros and cons of using underscore templates?

Pros:

- Ability to include logic in templates that are evaluated at runtime.
- Templating on even older browsers.

Cons:

- It's not reactive like KnockoutJS (although, you can use the two frameworks together).
- No automatic HTML escape (must use `<%-` for that).
- No translation.

Describe how underscore templates are used in Magento together with the text RequireJS plugin.

Magento uses the `text! RequireJS` plugin (prefixed to the template's path ID) to load HTML files that are executed with the `_.template()` method (usually indirectly via `mage/template.js`). This means chunks of HTML are loaded asynchronously and executed with a data object for context.

A good example of this on the front end is [breadcrumbs.html](#) (`Magento_Theme/web/templates/breadcrumbs.html`). The HTML template is loaded asynchronously

and rendered in the context of an array of breadcrumbs (not edible). Since the breadcrumbs don't change after the page is loaded, there is not a specific need for reactive data handling.

1.3 DEMONSTRATE UNDERSTANDING OF JQUERY UI WIDGETS

Demonstrate understanding of the jQuery framework [sic] and its role in the Magento JavaScript framework



AUTHOR NOTE

Technically, jQuery is a "library."

What is a jQuery library?

jQuery is a large library of Javascript tools with the ubiquitous jQuery object, `$`, being the linchpin. It quickly became immensely popular after its release in 2006, because of the consistent API it provided between browsers. At that time, developing Javascript to be cross-browser compatible was a complicated process because various browsers used complete disparate techniques for performing tasks.

XHR requests (commonly known as "AJAX") are a great example of that. Implementing them required long, complex methods with varying calls based on the browser. jQuery solved that problem by providing a single interface: `$.ajax` (plus its siblings like `$.get`). When browsers started developing and implementing

standards (some likely heavily influenced by jQuery's simplicity), the cross-browser differences dwindled to be much more manageable. Now, we have a well-supported `fetch()` call, for instance, to handle our XHR needs. jQuery has been the most popular Javascript library and still is widely used, although its popularity has declined markedly.

The common alias for the jQuery object, the `$`, is usually global, although Magento introduces some nuances with that. The `$('.some-selector')` method is often used to select a collection of elements. Methods can then be called on the collection to manipulate them or use them for some purpose. Methods can be added to the jQuery object via widgets—something to be discussed in more detail later.

What different components does it have (jQuery UI, jQuery events, and so on)?

There is an entire additional library that is built on jQuery: [jQuery UI](#). Its goal is to make certain types of functionality that are desired in many UIs available via a simple interface. Examples of that include tabs, accordions, and drag and drop. It is built on jQuery because it uses the jQuery object and DOM manipulation. In simpler terms, jQuery UI makes jQuery more powerful.

jQuery contains an [entire event system](#). Most of that has been eclipsed by the native, well supported `.addEventListener()` but jQuery's are still frequent in the core and important to be familiar with should the need arise to update one. There are some differences as well. Many of jQuery's event listeners can be attached using a method name that corresponds to the event. For instance, jQuery has a `.click()` method. The native equivalent is `.addEventListener('click')`. A significant aspect to jQuery's event listeners

is that they work regardless of whether it is a single element or a collection. Adding an event listener works like this: `$('.button').click(/* callback */) (all elements with the .button class would trigger this event listener).`

In addition to the DOM events, jQuery has [global AJAX event listeners](#). These over-achieving methods can be obscure when debugging because they perform significant actions in code far from the place where the request is sent. Magento core uses these event listeners to handle various tasks. For instance, in `lib/web/mage/backend/bootstrap.js`, all requests are listened to. If one comes back with a JSON object containing an `ajaxRedirect` key, the page is redirected—automagically. It also injects the `form_key` if making a JSON type request.

In Chrome DevTools, you can see the names of these AJAX events by selecting a DOM element—such as `body`. Select the “Event Listeners” from the group of tabs that contains “Styles” and “Computed.” There, if present, they will appear like: `ajax[Type] (ajaxComplete)`. This will lead you to the place in the jQuery library where the event is triggered. You can place a breakpoint in that function and step down to find the global event listener.

How does Magento use it?

Magento uses jQuery in a large percentage of its Javascript modules. It is imported with the `jquery` (all lowercase) module key. The `$` is used in the function declaration for the RequireJS module. Magento uses a wide range jQuery tooling in its core code: it’s used for DOM manipulation, XHR requests, widgets, and generic methods. It is important to always use jQuery with RequireJS. Using it without RequireJS makes it likely that a different jQuery object will be used, causing problems related to jQuery widgets.



AUTHOR NOTE

It is our opinion that like UnderscoreJS, there's little (if any) reason to use jQuery if your targeted browsers contain native APIs that do the same thing. We venture to offer the recommendation to obtain the habit of using native APIs when available and would reference [this helpful website](#) as a resource when doing custom development.

1.4 DEMONSTRATE UNDERSTANDING OF KNOCKOUTJS

Describe key KnockoutJS concepts

KnockoutJS is a library that facilitates a Model-View-ViewModel technique (if that sounded more like a Java term than Javascript, you're not alone). One of Knockout's most important aspects is binding data to HTML elements. Any time the data updates, the elements also update. An application then has a data structure ("Model") and templates ("View") with KnockoutJS assisting in linking the two ("ViewModel"). Elements are tied to data with a `data-bind` attribute. Overall, KnockoutJS feels like a precursor to frameworks like VueJS or React.

And, yes, Knockout supports IE 6.

Describe the architecture of the Knockout library: MVVC concept, observables, bindings.

MVVC is short for Model-View-ViewModel and is a [programming paradigm](#). The ViewModel sits in between the View and the Model, effectively connecting (binding) them. Knockout has four hallmark features—five if you count IE 6 support—

declarative bindings, automatic UI refresh, dependency tracking, and templating. Declarative bindings refer to the ``data-bind`` attributes that are added to elements in order to associate them with data. This binding is two-directional in some cases: for an input, if its value changes, it can update the model, or if the model updates, it can update the input. In Magento, ViewModels are Javascript modules. In the module, methods can be used to provide, assemble, or transform various pieces of data before being displayed. This file is an example: [Magento_Ui/js/form/element/wysiwyg.js](#). The [associated template](#) binds the `content` key (initialized in `defaults` of the Javascript module) as HTML.

With observables, KnockoutJS tracks any changes that occur in the underlying data. It then updates the View. Consider the classic technique of: `element.textContent = newValue`. With KnockoutJS and `knockout-es5` library, assuming that element has an attribute, `data-bind="text: dataKey"`, it will simply update the value.

Knockout has a number of different bindings available ([refer to the documentation to see](#) a more complete list), and Magento adds a number as well. Various bindings handle different interactions between the data and the element. A few examples:

- `text`: binds a value as the text content of an element. Would escape HTML as a result.
- `visible` if the value is false, the element is hidden. `if` has the same end result but actually removes the element's content.
- `foreach` loops over an array and binds each item to the corresponding array item. You probably aren't surprised to learn that this is especially useful for rendering lists and tables.
- `click`: adds an event handler. If you guessed it was for the click event, you are correct. It can bind to a method, or you could assign a value in the attribute itself: `data-bind="click: total = timer + 1"`.

Demonstrate understanding of knockout templates

What is the main concept of knockout templates?

KnockoutJS templates provide client-side rendering of dynamic data. In Magento, they are usually put in .html files and requested asynchronously, but they can be added directly to the body if the `scope` binding is used. The templates contain elements with `data-bind` attributes, or their [Magento equivalents](#), and update whenever the underlying data is changed.

Consider the catalog product grid in the admin panel: when a manager clicks the "next page" button to see more results, a new set of objects need to be displayed. Instead of reloading the entire page, the visible list re-renders due to the change in the data.

What are the pros and cons of knockout templates?

Pros:

- Facilitates dynamic data binding
- Links the UI more tightly and logically to the underlying data set reducing brittle DOM query and update mechanisms

Cons:

- Potential loss of SEO. Because Javascript must be executed in order for the content to render, it is possible that some search engine bots would miss the data. Search engine bots are growing in their ability to run Javascript and may not be an issue for much longer. It is not a problem for areas that require authentication to access either, like customer or admin areas.
- No automatic HTML escape

- Data-bind attributes can get unwieldy
- Delayed rendering (in some cases). Data must be loaded and available before it can be rendered.

Compare knockout templates with underscore JavaScript templates.

- Knockout templates automatically update when the underlying data changes.
- Underscore templates are rendered based on the data only when the string is passed to the `template()` function. No further connection to the underlying model is maintained.
- Knockout uses `data-bind` attributes.
- Underscore uses "ERB-style delimiters." ERB stands for "Embedded Ruby" (wait, what?).

Demonstrate understanding of the knockout-es5 library

Typically, to create an [observable](#), this code is used: `ko.observable(varName)`. Further, to access the value, it is necessary to call it like a function: `varName()`. Magento includes [Knockout-es5](#), a library that simplifies tracking observables a bit, and uses it as the primary means of initiating an observable. In many core Javascript modules, there is a section like the following:

```
initObservable: function () {  
    this._super()  
        .track([  
            'value',  
            'editing',  
            'customVisible',  
        ])  
}
```

This tracks specific properties on the parent object. It is itself an abstraction beyond `knockout-es5`, but it uses it all the same. The `knockout-es5` library uses a [simple](#) `ko.track` call for tracking an observable, but perhaps the more significant benefit is that it allows standard assignment and access of observed values (again, compared to Knockout's normal method-style of access). As a result, this works with `knockout-es5`:

```
var latestOrder = this.orders[this.orders.length - 1]; // Read a value
latestOrder.isShipped = true; // Write a value
```



HINT

There is a rudimentary but helpful Chrome [extension](#) for KnockoutJS.

2.

MAGENTO JAVASCRIPT BASICS



SWIFTOTTER
SOLUTIONS

2.1 DEMONSTRATE UNDERSTANDING OF THE MODULAR STRUCTURE OF MAGENTO

Demonstrate understanding of the JavaScript file organization in Magento



FURTHER RESOURCES:

Refer to [this DevDocs article](#).

What file structure is used to organize JavaScript modules?

Javascript modules are placed in a `web` folder of a [Magento component](#). This allows all of the component's files to be managed under a single directory.

Where does Magento locate a module's JavaScript file?

Javascript files are placed in `[Vendor]/[Module]/view/[area]/web/js`.

Areas are described [in more detail in the DevDocs](#) and relate to the section of the store that is being rendered. The most common ones are `adminhtml` and `frontend`. `base` is available to all areas. Javascript placed in an area should only be used in that area. Javascript files within `view/[area]/web/js` should be put within directories to provide logical groups.

If a theme overrides a Javascript file, that version of the file will be loaded. It is *critically important* to understand Magento's fallback system. As a rudimentary example, the following idea demonstrates the fallback system where the earlier

items in the following list are overridden by subsequent items (assuming a frontend area):

- Module_Name/base (area)
- Module_Name/frontend (area)
- base/theme_name
- frontend/parent_theme
- frontend/theme_name

Notice how a module's assets are overridden by themes and a more specific area overrides a less specific area. We want to stress that the preceding example does **not** cover all factors of inheritance and suggest you pursue the following resources if you are unfamiliar with these concepts.

Where does Magento store the JavaScript library?



FURTHER RESOURCES:

- [Theme structure](#)
- [Theme inheritance](#)
- [Module Dependencies](#)
- Javascript file [location walkthrough](#)

Magento has a special folder with tons of libraries and files in it: `lib/web/`. It is also in `vendor/magento/magento2-base/lib/web/`, but copied from there so the primary use is `lib/web`. There's a ton of stuff in there, and it would be a good idea to at least take a brief look: `lib/web/`.

One file that is particularly interesting is `legacy-build.min.js`. It is a conglomeration of old Javascript files and libraries, including Prototype, minified into one super file. It is used to bridge old, Magento 1 code up to Magento 2 but comes at a high cost of massive file size.

Describe how static content is organized in Magento

Publicly accessible files are symlinked (developer mode) or copied via CLI (production mode) to the `pub/static/` directory. From there, files are grouped by area, theme, locale, and module name in the form of `Vendor_Module` (like `Magento_Ui`).

How does Magento expose a module's static content to the web requests?

The files in the `view/[area]/web` directory are copied to the `pub/static` folder (symlinked in developer mode). For instance, a source Javascript file is placed in `app/code/SwiftOtter/Theme/view/frontend/web/js/hero.js`. This file is then referenced with `SwiftOtter_Theme/js/hero`. In developer mode, that file is symlinked to its end location in `pub/static`. The full directory structure of the public file is: `pub/static/frontend/SwiftOtter/[Theme Name]/[Locale]/SwiftOtter_Theme/js/hero.js`. As a result, the identifier is the last part of the path and does not include the file extension (`.js`).

What are the different ways to deploy static content?

In Developer mode, the files are automatically symlinked, and no action is necessary. For production environments, the files must be copied: `bin/magento`

`setup:static-content:deploy`. This is a super-command. It processes assets for all the areas, enabled themes, and locales. As of Magento 2.2.x, this command is *influenced by the database* ([reference](#)). The main aspects related to the database are the minification and bundling of assets. You can set these [values via the CLI now](#), but see [this StackExchange answer](#) for more information.

2.2 DESCRIBE HOW TO USE JAVASCRIPT MODULES IN MAGENTO

Use `requirejs-config.js` files to create JavaScript customizations

The `requirejs-config.js` is an important part of Magento's Javascript ecosystem. All these files are collected and concatenated to form a single version that is sent to the browser and processed by RequireJS on the client side.



FURTHER RESOURCES:

[DevDocs guide](#)

There are a few common tasks you can accomplish by adding a `requirejs-config.js` to a custom extension:

Override (replace) a core Javascript module. The following configuration example would load the `SwiftOtter_Theme/js/google-analytics` Javascript module *instead* of the `Magento_GoogleAnalytics` one. This is useful when a mixin is

not feasible but a modification must be made. It can also be helpful when a change is so significant that it is more efficient to replace the module.

```
var config = {  
    map: {  
        '*': {  
            'Magento_GoogleAnalytics/js/google-analytics':  
                'SwiftOtter_Theme/js/google-analytics'  
        }  
    }  
};
```

Run Javascript on every page. If you want to run a module on every page of the site, you can include it as shown below:

```
var config = {  
    deps: [  
        "SwiftOtter_Theme/js/core/light-run"  
    ]  
}
```

How do you ensure that a module will be executed before other modules?

Remember that listing a dependency after another one does not enforce load order at all. It is necessary to explicitly define what modules must be loaded before a module can be executed. It is possible to declare these dependencies using the

`shim` property of `requirejs-config.js`. Note that this has the same effect as adding a module to the list of dependencies in a `define()` function, but allows you to control order without modifying the target module. If you have access to the `define()` function, it's better to just use it because it is easier to comprehend when working with it in the future.

This example loads a polyfill before the module:

```
var config = {
  'shim': {
    'MutationObserver': ['es6-collections'],
  }
}
```

How can an alias for a module be declared?

Just like so:

```
var config = {
  map: {
    '*': {
      uiElement: 'Magento_Ui/js/lib/core/element/element'
    }
  }
};
```

What is the purpose of `requirejs-config.js` callbacks?

According to RequireJS docs: “Useful when `require` is defined as a config object before `require.js` is loaded, and you want to specify a function to `require` after the configuration's `deps` array has been loaded.” Magento declares `require` as a config option, so a callback can be declared in order to execute a function in the module.

Describe different types of Magento JavaScript modules

Plain modules

These are regular ones. The sky's the limit, though, so you can use them for anything. Like other modules, call the `define` function and include a callback within it. This callback often returns another function. In fact, it **should return a callback** if you use it with the `data-mage-init` attribute or `text/x-magento-init` script tag. Here's an example:

```
define([/* ... dependencies ... */], function () {
    function handleWindow() {
        // ...
    }

    return function () {
        handleWindow();
        window.addEventListener('resize', handleWindow);
    }
});
```

jQuery UI widgets

Declares a jQuery UI widget which can be used elsewhere. Always return the newly created widget as shown in the following example:

```
define(['jquery', /* ... */], function ($) {
    $.widget('mage.modal', {
        options: {
            // default options
            // options passed into the widget override these
        },

        /* Public Method */
        setTitle: function() {},

        /* Private methods being with _ (underscore) */
        _setKeyListener: function() {}
    });

    return $.mage.modal;
});
```

UiComponents

Javascript modules that are part of UI Components extend `uiElement` (`Magento_Ui/js/lib/core/element/element`). Carefully consider the following example:

```
define(['uiElement', /* ... */], function(Element) {
    'use strict';

    return Element.extend({
        // like jQuery "options." Can be overridden on initialization.
        // In Magento, these can ultimately be provided, or overridden,
        // from the server with XML or PHP. defaults: {
            // UI Component connections are discussed in
            // further detail later
            links: {
                // $.provider is equivalent to this.provider
                value: '${ $.provider }:${ $.dataScope }'
            }
        },

        // method is accessible in the associated
        // KnockoutJS template
        hasService: function () {},

        // 1 of 6 `init[...]` methods which can be
        // overridden and used for setup.
        initObservable: function () {
            this._super()
                .observe('disabled visible value');
        }
    });
})
```

Note that there are a number of different modules that extend `uiElement`. `uiCollection` is a common one and, as the name implies, facilitates dynamic lists. All UI Component Javascript modules follow the basic pattern of extending a base class with an object containing a `defaults` object and a number of functions. For one thing, the base class handles the template. This is why the methods (and properties of the `defaults` object) can be called from the template. We'll cover this in more detail later because it's complex.

2.3 DEMONSTRATE ABILITY TO EXECUTE JAVASCRIPT MODULES

There are a number of ways to load Javascript. Choosing the appropriate method for your use-case necessitates understanding each technique. A couple things to remember with the `data-mage-init` attribute and `text/x-magento-init` script:

- They provide a good way to pass server-generated configuration (not customer-specific data, though) into Javascript modules.
- They typically do not execute quickly after the page loads.
- Both require a significant amount of Javascript to load and execute before they are processed, loaded, and executed. While this often occurs at an acceptable speed, depending on the module's purpose, it may be an issue that users notice.

Demonstrate the ability to use the `data-mage-init` attribute to run JavaScript modules

The `data-mage-init` is perfect for succinct initialization of modules. The element which contains the attribute and the Javascript object are passed into the callback that the module is expected to return—this callback can also be a jQuery widget—

either are valid. The following is an example taken from the `state.phtml` file (located in [Magento_LayeredNavigation](#))

```
<div data-mage-init='{ "collapsible": {"openedState": "active",
"collapsible": true, "active": false } }'>
    <!-- ... -->
</div>
```

This use is a bit nuanced. The first `"collapsible"` is a reference to a particular module. This module ID is an alias to `mage/collapsible`. Files in `mage/` are in the `lib/web/mage` directory. If we look there, you will find a `collapsible.js` module. This module declares a jQuery widget. It is initialized with the `<div/>` being the targeted element and the configuration within the object as the options: `{ "openedState": "active", "collapsible": true, "active": false }`.

Here is another example:

```
<div data-mage-init='{ "SwiftOtter_Components/js/modal": <?=$block->getConfig(); ?>}' />
```

This initializes a component with the following structure:


```
define([/* ... */], function() {  
  
    return function(config = {}, element) {  
        // ...  
    }  
})
```

What is the purpose and syntax of the `data-mage-init` attribute?

The purpose is to easily initialize Javascript modules asynchronously. It serves as a progressive enhancement of elements already in the DOM. The syntax is an object containing module IDs as keys and configuration objects as values.

How is it used to execute JavaScript modules?

Magento has a Javascript module (`lib/web/mage/apply/main.js`) that finds elements with the attribute and uses the configuration in it to execute modules. All of this happens on the client side and at runtime; as such, there is a performance trade-off.

Demonstrate the ability to use `text/x-magento-init` scripts to execute JavaScript modules

This is the heavy hitter of Magento Javascript framework and is ultimately used for many things, including initialization of UI Components. The same module that runs the `data-mage-init` picks up and executes these script tags. Note that browsers

skip script tags with the type of `text`, so Magento can safely use them for storing configuration objects. The object within them is similar to the `data-mage-init` attribute with some important differences.

The initial key is a *selector* and its value (on the right side of the colon), is an object with the module ID and its configuration data. Magento will use the selector to query those elements and will execute the callback function for every *single* element it finds that matches the selector. For instance, if “table” is the key, Magento will load the component (once) and execute the callback for each `<table>` found on the page with that element as the second argument and the module’s component configuration as the first one.

Let’s look at some code:

```
<script type="text/x-magento-init">
  {
    ".hero": {
      "SwiftOtter_Theme/js/hero": {
        // ... options that would be passed into the
        // callback for every element
      }
    }
  }
</script>
```

This is pretty simple, right? It gets more complicated from here, but this minimal setup covers quite a few situations and serves as the foundation for more complicated renderings.

In this case, the hero module that was loaded has this structure:

```
define(['underscore'], function(_) {
    return function(config, heroElement) {
        // ...
    };
});
```

Remember that the config comes first, and the element is second.

What is the purpose and syntax of the `text/x-magento-init` script tag?

The `text/x-magento-init` script provides flexibility in initializing Javascript modules and tying them to elements. It is also used for bootstrapping UI Components. One of its core aspects is the ability to pass server-generated configuration to the front end—a task common to Magento.

The syntax is quite loose: outer curly braces (an object) that surround necessary configuration. The keys are element selectors, with one exception. If a `*` (asterisk) is used, the module is run once and not bound to any specific element. Objects are used for values there as well: keys are the module ID to load, and the value is passed as an argument to that module. This is often an object and can be as complex as needed.

What is the difference between the `text/x-magento-init` and the `data-mage-init` methods of JavaScript module execution?

The `text/x-magento-init` has few boundaries while the `data-mage-init` only applies to a single element. For `text/x-magento-init`, an object defines all information necessary to bootstrap modules and provides links to the elements they relate to. The `data-mage-init` targets one element and as a result, provides a slightly simpler interface for initializing modules.

Describe advanced methods of executing JavaScript modules

How do you execute a JavaScript module in an AJAX response and in dynamic code stored in a string?

There are a couple ways to do this. The first step will be to glean the Javascript code from the response or string. Sometimes it will come in the form of a static block that has a `<script/>` tag in it which must be separated from the rest of the response. Then the raw string of Javascript can be passed to the `eval()` function, or better `Function()`. Another option is to inject it into the `<head>`. If you do choose to use `Function()` or `eval()`, read and understand the downsides related to security and performance. While they are not [“bad” in and of themselves](#), they introduce risks that are important to understand first and reasonably mitigate.

If the goal is to process `data-mage-init` or `text/x-magento-init`, there are other ways to do it as detailed later, but briefly, either fire the `contentUpdated` jQuery event on the body, or re-execute `mage/apply/main`.

2.4 DESCRIBE JQUERY UI WIDGETS IN MAGENTO



FURTHER RESOURCES:

Refer to [Magento DevDocs](#) for information on some of the widgets available.

Describe how Magento uses jQuery widgets, and demonstrate an understanding of the `$.mage` object

They are frequently used in the `data-mage-init` attributes.

`$.mage` serves as a namespace for [Magento widgets](#). Magento uses the [widget factory](#).

What is the role of jQuery widgets in Magento?

Magento uses jQuery widgets to implement client-side functionality to a wide range of elements across the site. One particular benefit is their extensibility which allows third-party developers to override aspects of them.

What are typical widgets?



FURTHER RESOURCES:

Refer to the DevDocs for a [list of widgets](#).

Here are a few of the widgets available and examples of their use:

- [Tabs](#) ([Example Use](#))
- [Menu](#) ([Example Use](#))
- [Sticky](#) ([Example Use](#))
- [Toggle](#) ([Example Use](#))

It is important to take some time to review the various widgets' functionality and familiarize yourself with their different uses. For instance, consider three widgets that perform a similar job of hiding and showing segments of content: tabs, collapsible, and accordion. They each have nuances:

- [Tabs](#) show only one segment of content at a time. Selecting a new “tab” will cause others in the same group to disappear and the new one to display alone.
- [Collapsible](#) is a single element.
- [Accordion](#) is many collapsible sections in one group. It differs from tabs by allowing more than one to be open. It has settings that allow fine-tuning of this as well. One of such would be the `active` setting: an array or string of integers that can be used as indexes to show a number of the panels in the accordion when it loads.

The Alert, Confirmation, Prompt, and DropDown widgets are another case where there are multiple widgets that may appear similar on the surface but have important differences.

How are Magento jQuery widget modules structured?

Widgets are declared with calling jQuery's widget function like: `$.widget('mage.widget_name', { /* ... everything else */ })`. They contain an `options` key with an object of settings that are available in the widget. Practical defaults are declared to minimize boilerplate elsewhere in the app. After the options object,

functions comprise the bulk of the widget with “private” functions starting with an underscore (note that they can still be extended even when “private”), and public methods without.

Following is an example of a widget declaration. Note the `_create()` method which is used to set up the widget. There [are other methods](#) associated with the widget factory that are available like `_setOption()`.

```
$.widget('mage.list', {
    options: {
        template: '[data-role=item]',
        templateWrapper: null,
        destinationSelector: '[data-role=container]',
        itemCount: 0
    },
    _create: function () {
        this.options.itemCount = this.options.itemIndex = 0;
        this.element
            .addClass('list-widget');
    },
});
```

Describe how Magento executes jQuery widgets

jQuery widgets are often executed through the template system as described in the next section. However, another common use is to include them as a dependency in the RequireJS module. The `mage/validation` widget is perhaps one of the most frequent uses in this manner. They are appended to the list of other dependencies

but do not need to be added to the callback function's definition because they are accessed through the jQuery object.

Consider the following example based on the [Magento_Checkout/js/view/form/element/email](#) module and note how there is no `validation` item in the function declaration:

```
define([
    'jquery',
    'uiComponent',
    'ko',
    'Magento_Customer/js/model/customer',
    'Magento_Customer/js/action/check-email-availability',
    'Magento_Customer/js/action/login',
    'mage/validation'
], function ($, Component, ko, customer, checkEmailAvailability,
    loginAction) {
    /* ... */

    function validateEmail() {
        var loginFormSelector = 'form[data-role=email-with-possible-login]',
            usernameSelector = loginFormSelector + ' input[name=username]',
            loginForm = $(loginFormSelector),
            validator;

        loginForm.validation();
    }
});
```

```

        validator = loginForm.validate();

        return validator.check(usernameSelector);
    }
});

```

How are Magento jQuery widgets executed with `data-mage-init` and `text/x-magento-init`?

The widget is initialized on the element that has the `data-mage-init` attribute or is targeted in `text/x-magento-init` script. The module that contains the widget is requested and after loading, the result is tested: if it is a function, it's called like a function, but if it is a jQuery widget, it's executed like that. You can see the action in the `init()` function of `apply/main.js`.

2.5 DEMONSTRATE ABILITY TO CUSTOMIZE JAVASCRIPT MODULES

Use Magento JavaScript mixins for customizations

Mixins are to Javascript as plugins are to PHP in Magento. The logic is that you can add actions before, after, or instead of core functions. This allows you to manipulate core Javascript without always needing to override the entire file. The benefit is you have to write and maintain less code. They are quite easy to use as long as the core

is facilitating. There are some places where things get weird, but they can still be used.



FURTHER RESOURCES:

This [DevDocs article](#) will get you up to speed on the specifics of building mixins

Describe advantages and limitations of using mixins.

Mixins allow you to change functionality in other Javascript without needing to override the entire file. This, in turn, makes upgrading the core and other modules easier because your code does not replace the entire file. Obviously, there could be a change in the module that your mixin depends on, but fixing that is usually much easier than attempting to upgrade your override.

Conversely, because the mixin is an outside entity, it can only do so much. The following is an example of something that works around a problem where various functions were not very accessible from a mixin:

```
define(function () {  
    var original;  
  
    function getSettingOverride(mode) {var output = original.  
        call(this, mode);  
  
        output['theme_advanced_buttons1'] =  
            output['theme_advanced_buttons1'] + ',styleselect';  
        return output;  
    }
```

```

    }

    return function (target) {
        if (typeof tinyMceWysiwygSetup.prototype ===
            'object' && tinyMceWysiwygSetup.prototype.getSettings) {
            original = tinyMceWysiwygSetup.prototype.getSettings;
            tinyMceWysiwygSetup.prototype.getSettings =
                getSettingOverride;
        }
        return target;
    };
});

```

The biggest limitation mixins have is related to the architecture of the code within the module you are targeting.

What are cases where mixins cannot be used?

It is not possible to add mixins to Javascript that is not included with RequireJS, including Javascript in the HTML.

Describe how to customize jQuery widgets in Magento

There are many options available on most of Magento's jQuery widgets; these can be defined when the widget is called. Beyond whatever is available there, overriding functions in the widget, or adding functions, can be done with mixins. If the jQuery widget must be completely overhauled, it is possible to replace the

entire widget; at that point, it may be better to create a completely custom widget and extend the core one.

How can a new method be added to a jQuery widget?

This should be done with a mixin. There is a great answer on StackExchange that details [the approach](#).

When adding or modifying a widget, the callback inside the module should return the widget. It is passed the original widget via a parameter. In effect, this callback returns a brand new widget; there's a great deal of control with that. This is where jQuery's widget factory is helpful: it will merge objects that are passed as arguments into it. For instance:

```
$.widget('mage.priceBundle', parentWidget, { /* object with new
methods */ } );
```

How can an existing method of a jQuery widget be overridden?

The answer is nearly the same as the last question and should be done with a mixin.

Here's an example of a module that adds functionality to the `_create()` method of a parent widget. These methods should have a `this._super()` call in order to call the parent function and can override "private" methods.

```
define(['jquery'], function($) {
    function limitOptions(index, list) {
        // ...
```

```

    }

    return function(widget) {
        $.widget('mage.priceBundle', widget, {
            _create: function() {
                this._super();

                this.element.find('.options-list').
                    each(limitOptions.bind(this));
            }
        });

        return $.mage.priceBundle;
    }
});

```

What is the difference in approach to customizing [sic] jQuery widgets compared to other Magento JavaScript module types?

jQuery widgets are customized by essentially creating a new version of the widget with the same name as the original one. By including the original in the widget factory method, it extends all functions and options that are not overridden. Other Magento Javascript modules are extended via merging a mixin object onto the original one.

3.

MAGENTO CORE JAVASCRIPT LIBRARY



3.1 DEMONSTRATE UNDERSTANDING OF THE MAGE LIBRARY

It seems fair to describe the `mage` library as the linchpin of Magento's Javascript framework and a point from which everything comes to life.

Describe different types of Magento JavaScript templates

There are a number of different templating systems available in Magento:

- Underscore
- jQuery (rarely used)
- Knockout
- [Kinda: ES6](#)

Magento pseudo ES6 [template] literals and underscore templates in Magento

The underlying problem is that Internet Explorer does not support template literals (also referred to as template strings). To workaround for that, while still allowing for the use of template literals, Magento uses UnderscoreJS to compile strings used as templates in non-compliant browsers. The action happens in `lib/web/mage/utils/template.js`. Magento detects native template literal support; if not present it falls back to UnderscoreJS for evaluating the string. UnderscoreJS allows for the specification of interpolation regex which Magento provides as equivalent of ES6: `${ [...] }`.

This remarkable system does come with a caveat: when using Magento's template literals, it is necessary to use regular quotes (') and not backticks (`).

These ES6-like template strings are commonly (or even primarily) used in the `defaults` object of UI Component's to specify dynamic variables in an otherwise static medium. We will be discussing them in more detail later, so you can look forward to that.

Describe how to use storage and cookies for JavaScript modules

You will be shocked to learn that Magento uses a jQuery cookie plugin for helping with cookies. Tasty comments aside, the [jQuery Storage API](#) is the primary abstraction for leveraging browser storage.

To use this feature, it is necessary to add both `jquery` **and** `jquery/jquery-storageapi` to the module's dependencies. Even though the Storage API is a jQuery plugin, it is not included in the global jQuery package. There is no need to add a parameter in the module function for the Storage API because it is accessed by the jQuery object: `$`.

Following is an abbreviated [example from the core](#):

```
define([
    'jquery',
    'uiComponent',
    'jquery/jquery-storageapi'
], function ($, Component) {

    return Component.extend({
        defaults: {
            cookieMessages: []
        },
    },
```

```

        initialize: function () {
            this._super();

            this.cookieMessages = $.cookieStorage.get('mage-messages');

            $.cookieStorage.set('mage-messages', '');
        }
    });
});

```

A few things to consider in the previous example. Note that the `jquery/jquery-storageapi` dependency does not have a corresponding parameter in the function declaration. Further, this module follows the UI Component pattern of extending the `uiComponent` (which is a module alias). The Storage API is accessed with `$.cookieStorage`. In this case, the module saves the message locally then clears them.



AUTHOR NOTE

Note: if the cookies are set to be [HTTP-Only](#), it's not possible to set them with Javascript (also, consider reading the security cautions related to using [document.cookie](#)).

How do you use cookies in the module?

This is largely described in the previous section: use `$.cookieStorage.get('mage-messages')`

How is `localStorage` used in Magento?

`window.localStorage` is a well-supported tool for [storage data client-side](#).

Magento uses it to store customer and app data. Customer data might relate to the wishlist, app data might have to do with checkout fields. The [jQuery Storage API plugin](#) is used for getting and setting the data. Some of the primary consumers, like the Customer and Catalog extensions have their own modules as an abstraction.

Demonstrate the ability to use the JavaScript translation framework



FURTHER RESOURCES:

Refer to the [DevDocs topic](#) on translation.

Magento 2 has a mature and thorough localization framework in place. Translating strings in Javascript is more complex than its PHP cousin. The foundation consists of `.csv` files that have the original string (English, in the core) on the left and the translated string on the right. Here is an example of a CSV line for a English to German translation:

```
"Welcome back!","Willkommen zurück!"
"Forgot Your Password?","Passwort vergessen!"
```

These Javascript translations for the user's locale end up in JSON format in local storage. They are then used to translate dynamic strings from Javascript modules and Knockout templates.

How are CSV translations exported to be available in JavaScript modules?

This question is a bit unclear, but we will provide insight into how strings are translated in Javascript. Ultimately, the answer is to use the `i18n:collect-phrases` [command](#). This command collects strings that meet the translation criteria (as described [here](#)) and saves them into CSV format for translation. The `.csv` language files provide a list of all strings that need to be translated—they can be sent to translators for processing.

The translated strings still need to make it to the frontend. Here is how that happens: there is a `.phtml` template (`Magento_Translation/view/base/templates/translate.phtml`) comprised of a RequireJS call for translation stuff from the server. It only fetches the strings for the current store, or, more specifically, the locale. The server responds with a JSON object of the strings which the module then persists to `localStorage`. The server actually saves the JSON content as well, in a `js-translation.json` file within the theme directory in `pub/static`.



FURTHER RESOURCES:

Be sure to [visit DevDocs](#) for more guidance on translating the strings.



AUTHOR NOTE

If you can't get some Javascript translations to work, delete `js-translation.json` in your theme's folder within `pub/static`.



AUTHOR NOTE

You can see what strings are available to Javascript two ways. First, you could open the `js-translation.json` file. Second, in Chrome DevTools, select Local Storage under the Application tab. After picking the correct domain, filter to `mage-translation-storage`.

How is a new translation phrase added using JavaScript methods?



FURTHER RESOURCES:

Do as [DevDocs](#) says.

Briefly, add `jquery` and `mage/translate` as dependencies; then `$.mage.__('Translate this string please')`.

Describe the capabilities of the JavaScript framework utils components

The `lib/web/mage/util/...` extends Underscore—your favorite library just got more powerful!

What are the different components available through the `mage/utils` module?

These utils can:

- Wrap and extend functions
- Interpolate template literals
- Manipulate strings, objects, and arrays
- Generate a unique ID
- Compare values

We suggest reviewing the contents of the `utils/` [directory](#).

Demonstrate the ability to use and customize the validation module

A great example of using the validation module is: `Magento_Contact/view/frontend/templates/form.phtml`. Refer to [this article](#) for a thorough look at using [the library](#). Briefly, include `data-mage-init='{ "validation": {} }'` on the `form` element that contains the elements to be validated. On the form fields, use the `data-validate` attribute with Javascript object-like syntax to define the validation constraints.

What is the architecture of the validation modules in Magento?

The validation super-module is `lib/web/mage/validation.js`, but there are two other important links: `lib/web/jquery/jquery.validate.js` (of course!) and `lib/web/mage/validation/validation.js`. The last is the one used as an entry point. It includes `mage/validation`, which, in turn, includes the [jQuery validation library](#). The `mage/validation` module contains a `rules` variable with a large object containing many validation criterion: like `max-words` (fun fact, this one appears to be there for your enjoyment so do use it—the core doesn't). Also, if you are building a car parts store for the US, there's a `vinUS` validator.

The `rules` object is iterated and each one is used to extend jQuery's validation library.

How can a custom validation rule be added?



HINT

There's a great [answer here](#).

The addition involves creating a mixin for the `mage/validation` module and using `addMethod()` for the validator:

```
define(['jquery'], function($) {
    "use strict";

    return function() {
        $.validator.addMethod(
            'validate-example',
            function() {/* validation logic */},
            $.mage.__('Your validation error message')
        )
    }
});
```

How can an existing rule be customized?

Like the previous example, create a mixin for `mage/validation` and use `addMethod()` with the exact name of the validator to override. Copy and update the logic inside the function that determines whether or not the value passes.

3.2 DEMONSTRATE UNDERSTANDING OF MAGE WIDGETS



FURTHER RESOURCES:

We recommend that you [review DevDocs](#) for the widgets.

Describe the collapsible jQuery widgets in the Magento JavaScript library and how to use them

The `collapsible` [widget](#) hides content that follows a header until the heading is clicked. It then expands to show the content:

Closed:	Shopping Options	Shopping Options
	CLIMATE ▼	CLIMATE ^
		All-Weather (9)
		Cool (11)
		Indoor (9)
		Spring (11)
		Windy (9)

One way to use the collapsible widget is to add our trusty friend `data-mage-init` to an element that surrounds both the heading and content. It is used in this manner in the Luma theme's `state.phtml` template (`vendor/magento/theme-frontend-luma/Magento_LayeredNavigation/templates/layer/state.phtml`). This is the attribute from the file: `data-mage-init = '{"collapsible":{"openedState": "active", "collapsible": true, "active": false }}'`. The header has an important attribute: `data-role="title"`. The widget will also look for a related element with the attribute `data-role="content"`. If not found, it picks the element immediately following the title as the content. In the example provided, “openedState” is a class applied to the parent—this can be helpful if some secondary effect is desired when the section is open. The last two key/value pairs are redundant because they match the default settings.

Which collapsible widgets are available in Magento?

Collapsible widgets include: the collapsible widget as described above, [the accordion widget](#), and [tabs](#). Refer to the DevDocs to uncover the power of these widgets.

How do you use them?

The accordion is very similar to the collapsible widget. Use the `data-mage-init` attribute on an element with children that have `data-role` attributes. See [this question](#) for more [concrete examples](#). Demonstrate the ability to use the popup and modal widgets

These widgets follow common jQuery style and DevDocs has articles on them, so we will point you there:

- [Modal](#)
- [Alert](#)
- [Confirmation](#)
- [Prompt](#)

There are also wrappers for these widgets. They are found in `vendor/magento/module-ui/view/base/web/js/modal/` and handle some of the boilerplate for you.

How do you create a new popup, dialog, or modal with the Magento components?

The DevDocs references above contain examples of their direct use. To use the Magento wrappers, include the appropriate `Magento_Ui` module as a dependency. Then, call it as a function with whatever configuration argument is needed. For instance, let's use the following core module as an example: `vendor/magento/module-ui/view/base/web/js/grid/massactions.js`. This module handles changes on a grid where multiple items are manipulated together.

'`Magento_Ui/js/modal/alert`' is a dependency and a parameter in the module's callback function. In its `applyAction()` method, it is called if no items were selected. It is called with a configuration argument with a single value: `content`:

```
alert({
    content: this.noItemsMsg
});
```

Describe the different jQuery widgets in the Magento JavaScript library

Which other widgets are available in the Magento JavaScript library?

Other widgets we have not specifically referenced include:

- [Calendar](#)
- [DropDownDialog](#) (essentially, a more flexible `<select/>`)
- [Gallery](#)
- [Magnifier](#)
- [Loader](#)
- [Menu](#)
- [Navigation](#)
- [Quick Search](#)

How do you use them?

DevDocs contains a good amount of detail regarding their use. This usually involves the `text/x-magento-init` script tag, or selecting an element (or many) with jQuery and initializing a widget with it.

3.3 DEMONSTRATE THE ABILITY TO USE THE CUSTOMER-DATA MODULE

DevDocs has [an article](#) that describes much of the `customer-data` API. Be sure to read that; the following is only supplemental to it.

Demonstrate understanding of the customer-data module concept

The customer data module (`vendor/magento/module-customer/view/frontend/web/js/customer-data.js`) is the endpoint of a well-integrated and somewhat automated way to maintain customer state. It is able to determine when data becomes stale and update it. There is a complementary system on the server to provide the information.

What is private data?

It's data that is only applicable to one user. An example would be products that were recently viewed. These only are relevant and appropriate for the user who actually visited those product pages.

Why do we need to store information in the browser?

Data stored in the browser is immediately available—it's a great place to cache content. It is also a logical place to make data available that is only applicable to individual users.

What are performance considerations?

The data must still be synced with the server at some point, but that can be done as a background task. Beyond that, querying localStorage or even cookies is fast and does not necessitate a network connection.

Demonstrate understanding of how to use the customer-data module in customizations

How is the customer-data module structured?

The module uses global event listeners for AJAX requests. This makes sense because customer data would be manipulated through a request. If left unattended, the data stored locally would then be stale. If the module decides it is necessary to update the data, it will make a request of its own, after the original request has completed.

How is data accessed, invalidated, or set?

Access: require `Magento_Customer/js/customer-data`, then use its `get(sectionName)` method. The wishlist is a good example:

```
define(['uiComponent', 'Magento_Customer/js/customer-data'],
function (Component, customerData) {
    return Component.extend({
        initialize: function () {
            this._super();
            this.wishlist = customerData.get('wishlist');
        }
    });
});
```

Invalidation: future HTTP POST or PUT requests update `private_content_version`—the cookie which stores the version of the customer data. The module will then request a new version of the data.

Set: essentially the same as accessing the data—only use the `set(methodName, data)` method.

Describe how to use `sections.xml` to modify the information available through the customer-data module

This is [described here](#).

The typical format is the following:

```
<action name="catalog/product_compare/remove">
    <section name="compare-products"/>
</action>
```

The action name is the (beginning of) a request path to be matched. The section is the name of the section to update. If you do not include any `<section/>` nodes, all are invalidated.

How can a `sections.xml` file be used to add a new section and to modify the invalidation rules of an existing section?

The `sections.xml` files are merged, so you can use the same action name to update the sections. Beyond additions, a plugin for `\Magento\Customer\CustomerData\SectionConfigConverter::convert()` would provide a great deal of control.

How can a customization change the way existing sections work?

Changing URL structure could break the way the existing sections work. For instance, if for some reason, the URL used for adding a product to the cart was different than the default `checkout/cart/add`, the minicart would no longer update properly.

It is possible to add data to current sections by using a PHP plugin for the appropriate class because they must implement `\Magento\Customer\CustomerData\SectionSourceInterface`. `\Magento\Catalog\CustomerData\CompareProducts` is an example of this where `getSectionData()` could be intercepted.

4.

UI COMPONENTS



SWIFT OTTER
SOLUTIONS

4.1 DEMONSTRATE UNDERSTANDING OF KNOCKOUT CUSTOMIZATIONS

Describe Magento modifications to the Knockout template engine

The source files can be found in this directory: `vendor/magento/module-ui/view/base/web/js/lib/knockout/template/`. Magento has significant infrastructure around KnockoutJS, so there are aspects that will look unfamiliar to the KnockoutJS master. The following sections describe the modifications in detail.

Describe the remote template engine, template syntax, custom tags and attributes, and the rendering mechanism.

Remote Template Engine: KnockoutJS has a `template` binding but it expects all of the template as a string. To facilitate modularity in the platform, Magento added the ability to load templates from afar—the server. If it cannot find the requested template in the DOM, it will fetch it with RequireJS.

Template syntax, custom tags, and attributes: Magento uses a layer of abstraction for many of the KnockoutJS bindings. This comes in the form of custom tags (nodes) and attributes. Instead of `<div data-bind="template: templateUrl"> </div>`, it is possible to do: `<div template="templateUrl"> </div>`. There are many aliases available, and they do not all share a similar structure so we recommend you spend some time reviewing them.

A few examples:

- To bind a checkbox's state to a variable, either of the following is valid:
 - `<input type="checkbox" data-bind="checked: isChecked"/>`

- `<input type="checkbox" ko-checked="isChecked"/>`
- For an `if` statement, the following are valid:
 - `<!-- ko if: isVisible--><!-- /ko -->`
 - `<if args="isVisible"></if>`
 - `<div data-bind="if: isVisible"></div>`
 - `<div if="isVisible"></div>`
- Or, for a negative `if`:
 - `<!-- ko ifnot: isVisible--><!-- /ko -->`
 - `<ifnot args="isVisible"></ifnot>`
- For a template:
 - `<div data-bind="template: templateUrl"></div>`
 - `<div template="templateUrl"></div>`



FURTHER RESOURCES:

[DevDocs Topic](#)

History Note: officially, Magento says this custom syntax is to make templates simpler to write and easier to read. Interestingly, according to a source familiar with the development of Magento 2, the original idea was to build a system where KnockoutJS could be replaced. Given the functionality that was built to facilitate the custom template syntax, this make sense. It sounds like that goal was given up on, though, due to the permeation of KnockoutJS through the platform and perhaps disparity between it and the newer frameworks available.

Rendering mechanism: Magento has overridden the core KnockoutJS template engine, noting that it: "Caches templates after it's unique name and renders it once." The file that does this is: `vendor/magento/module-ui/view/base/web/js/lib/knockout/template/engine.js`.

Demonstrate the ability to use the custom Knockout bindings provided by Magento

Where can a full list of custom bindings be found?

You can find the [list here](#).

What are they used for?

Refer to the DevDocs for specifics; they cover a wide range of functionality. `collapsible` is an example. It is similar to the jQuery widget with regards to configuration options.

What alternatives are there?

Some of the bindings can be used as attributes or nodes, as described in the recent section on custom tags and attributes. Be sure to familiarize yourself with the various options. You can also create your own bindings, nodes, and attributes (or binding aliases). To create your own binding, look at the [scope binding source](#). `ko.bindingHandlers.scope` takes an object with various methods to perform the logic. The [renderer](#) is a dependency, and a node is created by calling the `addNode()` method. If the node name is different than the binding, pass an object in as a second argument to reference the binding (`{ binding: example`

}). To create an attribute call the `addAttribute()` method in a similar way to `addNode()`.

Describe the Knockout scope binding

First, a note on context in Knockout: in the template, the current binding context is equivalent to `this` in plain Javascript. The context can contain methods or static values. Other variables are added to the current context in Magento but are referenced by prepending the name and a period to the value: `$parent.value`.

Consider this over-simplified example:

```
var scope = {
    contents: 'Display Me.',
    foo: function() {
        alert('Variety is the spice of life.')
    }
}

<div data-bind="text: contents, afterRender: foo">
```

The scope binding may be the most important binding to understand due to Magento's use of it. The scope binding sets up the connection between a template and its Knockout context by evaluating descendant nodes in the scope of an object found in the Ui Registry. **In Magento's Javascript framework, KnockoutJS templates must have a parent with a scope binding.** Not all HTML templates will have a scope binding, but this is because they are requested within a different template that would have context.

What is the purpose of the scope binding?

The scope binding provides a simple way to connect a Javascript module (specifically, one in the Ui Registry) to a template area. The value of the scope binding should be a string that identifies the item in the Ui Registry.

What Knockout problem does it solve?

It allows KnockoutJS to be initialized before it applies a ViewModel—a module that is requested asynchronously. There is a [good answer](#) on StackExchange that deals with some of [this](#).

How exactly does this binding work?

An identifier is provided to the `scope` binding. This is used to lookup the component in the Ui Registry (note, this is usually different, and shorter, than the component's RequireJS identifier). For instance, as seen in the example below, the Ui Registry identifier is “wishlist” while RequireJS uses “Magento_Wishlist/js/view/wishlist.” If the component is found in the Ui Registry, the associated module is loaded via RequireJS. The object in the module is used to populate a ViewModel on the nodes within the `scope` element.

The binding logic happens in: `vendor/magento/module-ui/view/base/web/js/lib/knockout/bindings/scope.js`

Take the following example for usage:

```

<li class="link wishlist" data-bind="scope: 'wishlist'">
    <a <?= $block->getLinkAttributes() ?><?=
        $block->escapeHtml($block->getLabel()) ?>
        <!-- ko if: wishlist().counter -->
        <span data-bind="text: wishlist().counter" class="counter
            qty"></span>
        <!-- /ko -->
    </a>
</li>
<script type="text/x-magento-init">
    {
        "*": {
            "Magento_Ui/js/core/app": {
                "components": {
                    "wishlist": {
                        "component": "Magento_Wishlist/js/view/
                            wishlist"
                    }
                }
            }
        }
    }
</script>

```

Demonstrate the ability to use the scope binding in customizations

How is the scope binding used?

The scope binding is used frequently because it is the way that the KnockoutJS context is bound to a template. The following contains an example where a child is loaded and bound to the child's UI Component's context: `vendor/magento/module-ui/view/base/web/templates/grid/sticky/sticky.html`. The `requestChild()` method loads a module from the Ui Registry with the applicable key. This is a good way to nest modules. The basic concept is:

```
<scope args="requestChild('listing_paging')" render="totalTpl"/>
```

We recommend experimenting with the `scope` binding if you are not already familiar with it. While the previous example has concrete configuration you can build based off of, create yours with the following general steps:

- Manually declare a script with the type of `text/x-magento-init`.
- Create a `components` object with one or more unique keys and assign it to the `Magento_Ui/js/core/app` module.
- The value of `components` should be an object which contains a `component` key with the RequireJS path to a module.
- Add a template with a `scope` binding that matches the unique key of its parent.

How do nested scopes work?

Each nested scope has a new binding scope as [described here](#). Parent scopes are accessible with `$parentContext` or `$parents` (an array of parents).

How can data of a parent scope be accessed from a child?

Use either `$parentContext` or `$parents` to access their context. Keep in mind that if structure changes and a custom module depends on a specific parent, it will cause trouble.

How can the scope binding be applied to HTML in Ajax responses?

There are two ways to do this after the response has been received: fire a `contentUpdated` event or manually run with `mage/apply/main`.

To fire `contentUpdated`: `jQuery('body').trigger('contentUpdated');` (`lib/web/mage/mage.js`). To manually run, add `mage/apply/main` as a dependency, and call `mage.apply()` after the content has loaded.

4.2 DEMONSTRATE UNDERSTANDING OF MAGENTO UI COMPONENTS



FURTHER RESOURCES:

Overview of [UI Components](#).

Describe how `uiComponents` are executed in Magento.

What is the difference in uiComponent execution compared to other JavaScript module types?

UI Component Javascript modules extend one of several abstract Javascript classes, and there are several methods that are called during initialization. The methods can be extended and used to set up functionality. UI Components also have templates associated with them and provide a KnockoutJS context.

What does it mean to "execute a uiComponent"?

Use configuration to load a template and Javascript module (plus supporting modules) and execute the Javascript in the module alongside rendering the template.

Why do we need the app component to execute uiComponents?

UI Components are configured through a structure of Javascript objects that define the various components. Many of these components can be cached and reused on the page. The app module is the entry point for the configuration.

What is the role of the layout component?

The layout component is loaded and executed from the app component. It loads the UI Components and saves them to the UI Registry. The entry point is the `run()` function. There are more details in the [DevDocs topic](#).

Describe the structure of a UiComponent

What is uiClass?

A very abstract class that introduces the basic architecture of UI Components.

`uiClass` is an alias for `Magento_Ui/js/lib/core/class`.



FURTHER RESOURCES:

Refer to [the DevDocs](#) for more information about its role.

How does it instantiate uiComponents?

The layout component initializes the UI Component through the `uiClass`. This occurs in the `initComponent()` method of the layout module where it calls `new Constr(...)` (presumably short for “Constructor” which is a reserved keyword in Javascript). This constructor function is returned from `uiClass` and handles some inheritance, while primarily calling the `initialize()` function. This, in turns, calls `initConfig()` which takes the `defaults` object, processes it (including getting the template from it and rendering that), and embeds it onto the module directly while parsing various keys (i.e. `uiComponent.defaults.key` to `uiComponent.key`). Each of the functions that have been overridden in the concrete UI Component are wrapped. This is what allows the extended function to be able to call `this._super()`.

How can existing component instances be accessed?

There is a component referred to as the UI Registry. It is a central aspect of UI Components and a tool for certain debugging tasks; it can be accessed in the following way from the console: `require('uiRegistry')`. It serves as a store for essentially all the modules in the UI Component system. For instance, open

the CMS Page Listing in the Magento admin panel and run the following in the Javascript console: `require('uiRegistry').get('cms_page_listing.cms_page_listing_data_source')`. It will return the listing's data source component instance.

The Registry is used from within components to access other components. Often template literals are used to form the reference to the other module, but after it has been processed, the plain, long form reference is used to look up the other module. This reference is a chain of UI Component names. The `'cms_page_listing.cms_page_listing_data_source'` reference is the UI Component named “cms_page_listing_data_source” which is inside the UI Component named: “cms_page_listing”. For an example of how template literals are used in this way, refer to the sample code regarding forms in section 4.4. One excerpt, though, is the following: `${$.parentName}.master_field:checked`. Here, we use the “variable” `$.parentName`, which would evaluate to `referral_form.dynamic_fieldset`, plus `master_field` to create the full reference of `referral_form.dynamic_fieldset.master_field`, targeting a very specific UI Component. The word after the colon (`:`) is a property of the module object.

How can a uiComponent be modified?

UI Components' logic or other aspects of Javascript can be modified using mixins. They shine in this way as long as the functions in the module are part of the primary module object. Configuration, conversely, should be modified in other ways. This can be done in the [component's XML declaration](#) or with [a DataModifier](#). All of this is output into Javascript objects on the page and is used when the module is initialized: it always takes precedence of the `defaults`.

How do you extend an existing uiComponent?

Create a Javascript module similar to the one shown below. Instead of the dependency on `uiElement`, use whatever UI Component is to be extended. Override and add functions and properties as necessary.

```
define([
    'uiElement'
], function (Element) {
    'use strict';

    return Element.extend({
        defaults: {
            // not required—just an example here
            links: {
                value: '${ $.provider }:${ $.dataScope }'
            }
        },

        // example function available in KnockoutJS template
        hasAddons: function () {
            return false;
        },

        // example observable initialization function
        initObservable: function () {
            this._super()
                .observe('disabled visible value');
        }
    });
});
```

```
        return this;
    }
});
});
```

Again, we will take a moment to point out the template literal in the previous snippet. In this case, `$.provider` would evaluate to a string that can be used to perform a module lookup in the UI Registry. Then, there is the all-important colon (`:`) separator, and then another template literal that is evaluated to target a specific property.

What is the role of the `uiElement` and `uiCollection` modules?

Both are good modules to extend from when creating a UI Component: `uiElement` if the component is based on a single element and `uiCollection` if it is a collection of items. They both directly extend `uiClass` and provide additional functionality.

- `uiElement`
- `uiCollection`

Demonstrate the ability to create a `uiComponent` with its own data, or operate with data of existing `uiComponents`

How does a uiComponent access the data it needs?

The role of a data source is to provide data to [UI Components](#). It is comprised of two parts: the server side Data Provider and a Javascript data provider module. On the server side, there is also a DataModifier, which allows data to be manipulated as it is passed through. The Javascript module is responsible for interacting with (some of the) data in a UI Component. Here is an example for a form: `vendor/magento/module-ui/view/base/web/js/form/provider.js`. The reason we note that it does not handle all data is because they are quite a long way from modern state management tools like Redux or Vuex. Data sources provide data from the server to the frontend.

What are the requirements for a subcomponent to provide data?

The module should extend `uiElement`. For example:

```
define(['uiElement'], function(Element) {
    return Element.extend({

    })
});
```

How can data be loaded by Ajax?

There is a URL endpoint that can be used to fetch chunks of data: `mui/index/render`. Loading data through that endpoint is common for listings (grids). The grid provider, `Magento_UI/js/grid/provider`, configures `Magento_Ui/js/grid/data-storage` for its `storageConfig`. The name

is slightly misleading because that module focuses on loading data. In the grid, whenever the state is changed (i.e. next page is clicked), a new set of data is requested: `var request = this.storage().getData(this.params, options);`. A request is initiated after this manner (%5D and %5B replaced for readability): `/mui/index/render/?namespace=product_listing&filters[placeholder]=true&paging[pageSize]=20&paging[current]=2&search=&sorting[field]=entityid&sorting[direction]=asc&isAjax=true`.

How can a component receive the data when it is loaded?

Add a callback for `.done()` on the request object returned from the storage component. This callback receives the data as an argument. Here is an example taken from the listing provider:

```
reload: function (options) {
    var request = this.storage().getData(this.params, options);

    request
        .done(this.onReload);
}

onReload: function (data) {
    this.setData(data)
        .trigger('reloaded');
}
```

Describe the process of sharing data between components

How can one uiComponent instance access data of another instance?

The native approach is to use one of: `imports`, `exports`, `links`, and `listens`. They are described in more detail on the [DevDocs site](#).

How can components communicate while taking into account their asynchronous nature?

A problem arises when a module that depends on another module loads first. Referencing a dependency when it hasn't loaded will result in an undefined object. There are two ways the core works around this: add a `deps` (preferable), or use a Knockout observable. The `deps` works in the same way that RequireJS dependencies do: it loads the child before executing the parent. The `deps` must be declared in the JSON configuration (although, often that would be via XML). Here is an example from: `vendor/magento/module-sales/view/adminhtml/ui_component/sales_order_shipment_grid.xml`.

```
<listing>
    <settings>
        <deps>
            <dep>sales_order_shipment_grid.sales_order_shipment_
                grid_data_source</dep>
        </deps>
    </settings>
</listing>
```


Using `deps`.

The “UI Component” way to use Knockout observables would be to set a property as an observable (with `tracks`), then use `imports` to obtain the data from the other source. Vinai Kopp proposes a better alternative of creating a module just for the purpose of tracking state. It would return an observable object. See [his video](#) for more details.

4.3 DEMONSTRATE THE ABILITY TO USE UI COMPONENTS

Describe the uiComponents lifecycle

What are the stages of uiComponent execution?

All the UI Component configuration that is handled with PHP and XML is processed and encoded as JSON during the page load. It is rendered in a `text/x-magento-init` script with the outer structure being:

```
<script type="text/x-magento-init">
    {
        "*": {
            "Magento_Ui/js/core/app": {
                "types": { /* ... */ },
                "components": { /* ... */ }
            }
        }
    }
</script>
```

The `Magento_Ui/js/core/app` is the entry point that must process the massive Javascript object. The `types` object is processed first and contains configuration information related to specific types of UI Components to be shared among components that extend them. The `components` object is where most of the configuration typically is held. Each component is loaded, with any dependencies loaded first. Configuration for the component is passed into its constructor method, and the module is added to the UI Registry for later use.

Now that the module has been spun up, it starts its work with the abstract `uiClass` processing the `defaults` object and running the various `init` methods.



RELATED READING

Check out [this article](#).

What is the role of the layout module, and how does it load components, children, and data?

The configuration object passed to `Layout` is processed by recursively loading the RequireJS module keys. Data provided to the object on the page is passed into the UI Component's constructor module.

What are the types of components it supports?

Any module that returns a function.

Demonstrate the ability to use uiComponents configuration to modify existing instances and create new instances

Let me make a critical point regarding UI Components: XML configuration permeates into Javascript modules and allows you to exert *tremendous* control over nearly any UI Component. Beyond XML, PHP `DataModifier` classes offer even more precise control. This is a difficult concept to explain because it requires some familiarity with the UI Component system to see the connection, but bookmark it in your mind for future work on UI Components. As an example, take a look at my [answer here](#). The asker is referring to something that is handled by a Javascript module, yet my answer entails only XML.

Describe the definitons.xml [sic] file and uiComponent instance XML files.

`definition.xml` (note our correct rendering of the file name) is a unique file that forms the basis of all UI Components. It cannot be directly changed but can be extended on a per-component basis. ([Source](#)). Be sure to spend some time going

through the file: it is also a helpful resource to refer back to when developing UI Components.

UI Component XML files are used to modify configuration for individual UI Components. These are merged into, but take precedence over, `definition.xml`. We recommend reading the [DevDocs topic](#) on configuration.

How can you modify an existing instance of a uiComponent using a configuration file?

Configuration XML files for any given UI Component are merged together. Take `product_listing.xml` for instance. While the primary file is in `Magento_Catalog`, there are other files in `Magento_CatalogSearch` and `Magento_Inventory`. Any other files created in the same directory structure will be merged in as well. In this example, that could be `Namespace_ModuleName/view/adminhtml/ui_component/product_list.xml`. To ensure a module you work on is merged after other modules, add those modules to your `etc/module.xml` [sequence tag](#).

To summarize the above question: create a file that matches the (1) directory structure and (2) schema structure of the file you are modifying and supply different values.

What is the role of the Magento layout in the uiComponent workflow?

UI Components can be placed onto a page with the `<uiComponent/>` [layout directive](#).

4.4 DEMONSTRATE UNDERSTANDING OF GRIDS AND FORMS

Customize existing grids and create new grids

How do you create a grid?

A grid is a very common pattern in use in Magento. We suggest that you implement one in a test environment to become familiar with the process. The initial grid is created with an XML file. Custom and dynamic functionality is added with Javascript.

The following is a very minimal example. The data source's class is a PHP class that has a CollectionFactory injected into its constructor.

```
<?xml version="1.0" encoding="UTF-8"?>

<listing xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xsi:noNamespaceSchemaLocation=
"urn:magento:module:Magento_Ui:etc/ui_configuration.xsd">

    <argument name="data" xsi:type="array">

        <item name="label" xsi:type="string" translate="true">Referral Listing</item>

        <item name="js_config" xsi:type="array">

            <item name="provider" xsi:type="string">referral_listing.referral_listing_data_source</item>

            <item name="namespace" xsi:type="string">referral_listing</item>

        </item>

    </argument>

    <settings>

        <deps>

            <dep>referral_listing.referral_listing_data_source</dep>

        </deps>

    </settings>

    <dataSource name="referral_listing_data_source" component="Magento_Ui/js/grid/provider">

        <settings>

            <storageConfig>

                <param name="indexField" xsi:type="string">entity_id</param>

            </storageConfig>

            <updateUrl path="mui/index/render"/>

        </settings>

        <dataProvider class="SwiftOtter\Example\Ui\Data\ListingProvider" name="referral_listing_data_source">

            <settings>

                <requestFieldName>id</requestFieldName>

                <primaryFieldName>entity_id</primaryFieldName>

            </settings>

        </dataProvider>

    </listing>
```

```
</dataSource>

<columns name="referral_listing_columns">

  <column name="name">

    <settings>

      <label translate="true">Name</label>

    </settings>

  </column>

</columns>

</listing>
```

How do you add an image column, standard validation, custom validation rules, and custom column types to a grid?

Image:

```
<column name="thumbnail" class="Magento\Catalog\Ui\Component\
Listing\Columns\Thumbnail" component="Magento_Ui/js/grid/columns/
thumbnail" sortOrder="20">
    <settings>
        <altField>name</altField>
        <hasPreview>1</hasPreview>
        <addField>true</addField>
        <label translate="true">Thumbnail</label>
        <sortable>false</sortable>
    </settings>
</column>
```

Standard Validation:

```
<column name="title">
    <settings>
        <filter>text</filter>
        <editor>
            <validation>
                <rule name="required-entry" xsi:type="boolean">true</rule>
            </validation>
        </editor>
    </settings>
</column>
```



```

        <editorType>text</editorType>
    </editor>
    <label translate="true">Title</label>
</settings>
</column>

```

Custom Validation:

Add a custom validator with the technique shown later in relation to adding custom validation for form elements in UI Components. Then, reference it as shown in the preceding example.

How do you modify existing grids?

As described above, place an XML file in the same relative directory as the one that contains the grid component. Match the schema and use different value or add applicable sections.

How do you customize the data loading process for a grid, including filters and sorting?

Manipulating data can often be done on the server side. In this case, a `DataModifier` is an [excellent tool](#). When customization should be done client-side, there are two options that should be chosen depending on the use: either create a mixin or custom Javascript module. Creating a mixin is a good option when the UI Component has a non-generic Javascript module assigned to it. An example of that is in `product_attribute_add_form.xml`: `component="Magento_Catalog/`

`js/components/new-attribute-form`". This module extends one of the generic UI Component Javascript modules and adds functionality. This concept would also apply for a provider module as well.

For instances where the UI Component uses a generic Javascript module, it may be a good time to create a custom Javascript module that extends the core module. Remember that it may not be clear in the UI Component's XML file what the Javascript module in use is. Everything extends from `definition.xml`, so the file reference can be found there. After creating the Javascript module, create the UI Component XML file in your module and assign your component to the source node.

Customize existing forms and create new forms

How do you create a form, a form with tabs, a form with groups of fields, a form with dynamic fields (when one field change will cause a change in another place)?

Even Javascript developers need to be familiar with XML when working with Magento. At this point, there is a stack of configuration necessary to create a form with tabs (or whatever) with UI Components. As a result, we recommend you carefully look through the following example and remember as much as possible. We have made notes throughout it. While it is long, we have worked to make this a minimal example that fills each of the operatives listed.

```
<form xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xsi:noNamespaceSchema
Location="urn:magento:module:Magento_Ui:etc/ui_configuration.xsd">

    <argument name="data" xsi:type="array">

        <item name="label" xsi:type="string" translate="true">Referral</item>

        <item name="js_config" xsi:type="array">

            <item name="provider" xsi:type="string">

                referral_form.referral_form_data_source</item>

            <item name="namespace" xsi:type="string">referral_form</item>

        </item>
    </argument>

    <settings>

        <layout>

            <!-- This is what makes the tabs go -->

            <navContainerName>left</navContainerName>

            <type>tabs</type>

        </layout>

        <deps>

            <!-- You do you. This is needed or else a weird,
            unhelpful Javascript error will be thrown. -->

            <dep>referral_form.referral_form_data_source</dep>

        </deps>

    </settings>

    <dataSource name="referral_form_data_source">

        <!-- All the regular dataSource stuff -->

        <argument name="data" xsi:type="array">

            <item name="js_config" xsi:type="array">

                <item name="component" xsi:type="string">Magento_Ui/js/form/
                provider</item>

            </item>

        </argument>

    </dataSource>

</form>
```

```

        </item>

    </argument>

    <dataProvider class="SwiftOtter\Example\Ui\DataProvider"
        name="referral_form_data_source">

        <argument name="data" xsi:type="array"/>

        <settings>

            <requestFieldName>id</requestFieldName>

            <primaryFieldName>entity_id</primaryFieldName>

        </settings>

    </dataProvider>

</dataSource>

<fieldset name="referral_fieldset">

    <settings>

        <label translate="true">General Info</label>

    </settings>

    <field name="entity_id" formElement="hidden">

        <!-- Useful for data that shouldn't change but needs
        to be sent with the payload when saving -->

        <settings>

            <label>Hidden Entity</label>

            <dataType>text</dataType>

            <visible>false</visible>

        </settings>

    </field>

    <field name="referral_field" formElement="input">

        <settings>

```

```
<label>Custom Field</label>

<validation>

    <!-- Any validation rule -->

    <rule name="validate-test"
        xsi:type="boolean">true</rule>

</validation>

</settings>

</field>

</fieldset>

<fieldset name="second_fieldset">

    <!-- Each <fieldset> is a new tab -->

    <settings>

        <label>Second Tab</label>

    </settings>

    <field name="simple_field" formElement="input">

        <settings>

            <label>Simple Input</label>

        </settings>

    </field>

</fieldset>

<fieldset name="dynamic_fieldset">

    <settings>

        <label>Dynamic Tab</label>

    </settings>
```

```
<field name="master_field" formElement="checkbox">

  <!-- Dynamic Field: Toggling this checkbox hides/
  shows the following field -->

  <settings>

    <label>Toggle Child</label>

  </settings>

</field>

<field name="dependent_field" formElement="input">

  <settings>

    <label>Dependent Field</label>

    <imports>

      <!-- All the Knockout observable goodies are
      already handled in the uiElement module -->

      <link name="visible">${$.parentName}.master_field:checked

    </link>

    </imports>

  </settings>

</field>

</fieldset>

</form>
```

How do you customize existing forms?

Do this the same way as you would a grid.

How do you add validation to fields, including custom validation rules?

See the `<validation/>` node in the section above. Adding a custom rule for a UI Component is similar to the jQuery style on the front end but **not** identical.

Create the validator module that follows this pattern. It is important to return the `validator` or nothing will work:

```
define(['jquery'], function($) {
    "use strict";

    return function(validator) {
        validator.addRule(
            'validate-example',
            function(value) { /* validation logic */},
            $.mage.__( 'Your validation error message' )
        );

        return validator;
    }
});
```

Create a mixin for either `adminhtml` or `base`:

```

var config = {
  config: {
    mixins: {
      'Magento_Ui/js/lib/validation/validator': {
        'SwiftOtter_Example/js/validator-trial': true
      },
    }
  }
};

```

How can you add a file upload field, an image field, and a custom field to a form?

File/Image Upload:

```

<field name="head_shortcut_icon" formElement="fileUploader">
  <settings>
    <notice translate="true">Allowed file types: ico, png, gif,
      jpg, jpeg, apng.</notice>
    <label translate="true">Favicon Icon</label>
    <componentType>fileUploader</componentType>
  </settings>
  <formElements>
    <fileUploader>
      <settings>
        <allowedExtensions>jpg jpeg gif png ico apng

```



```

        </allowedExtensions>
        <maxFileSize>2097152</maxFileSize>
        <uploaderConfig>
            <param xsi:type="string" name="url">theme/
                design_config_fileUploader/save</param>
        </uploaderConfig>
    </settings>
</fileUploader>
</formElements>
</field>

```

For custom fields, define a `elementTmpl` which references a `.html` file that contains the field you desire. This could be a completely custom field that you built, or, like our recent case, a different core field. For instance, we recently used Magento's fancy select field (with filterable results and the like) in place of the default select by simply setting the `elementTmpl` and `component`.

5. CHECKOUT



5.1 DEMONSTRATE UNDERSTANDING OF CHECKOUT ARCHITECTURE

The Magento checkout is built with UI Components but differs from the admin side in some of the implementation details. For instance, the configuration is handled via nested nodes in layout XML, the like of which many have never seen before. Just wait until you want to customize something: [it's jaw dropping](#). Fortunately, even though it is verbose and remarkable, it is not especially difficult, and it is very extensible (despite its infamous reputation).



FURTHER RESOURCES:

We will once again recommend the [terrific DevDocs](#) which contain a list of common customizations.

Describe key classes in checkout JavaScript: Actions, models, and views

What are actions, models, and views used for in checkout?

Navigate to the `vendor/magento/module-checkout/view/frontend/web/js` folder to see three primary subfolders: actions, models, and views. This seems to follow the MVC pattern that many developers are familiar with.

How does Magento store checkout data?

Temporary data is stored in localStorage, or in the cookies if localStorage is not supported. At the root, the `Magento_Customer/js/customer-data` module persists and loads the data. The `Magento_Checkout/js/checkout-data`

module provides a layer on top of that to allow a consistent interface to interact with data via methods. Beyond that, the `Magento_Checkout/js/model/checkout-data-resolver` handles a number of setup and other tasks related to the data. For example, when checkout starts, it creates a new shipping address, or loads the existing state if the user has already started. All of this is stored with the `mage-cache-storage` key in Local Storage. To see, open the Application tab in Chrome's DevTools. Open the Local Storage menu on the left sidebar and select the current domain. Select the `mage-cache-storage` item for a preview of the JSON.

What type of classes are used for loading/posting data to the server?

The Rest API provides the integration with the checkout front end and server. Model classes (Javascript modules) are primarily relegated the task of handling this transfer of data. However, a few action modules also perform similar functions.

How does a view file update current information?

Updated information is persisted through the `checkout-data` module.

Demonstrate the ability to use the checkout steps for debugging and customization

How do you add a new checkout step?



FURTHER RESOURCES:

Refer to the [DevDocs topic](#).

How do you modify the order of steps?

Change the sort order of the children of the `steps` node.

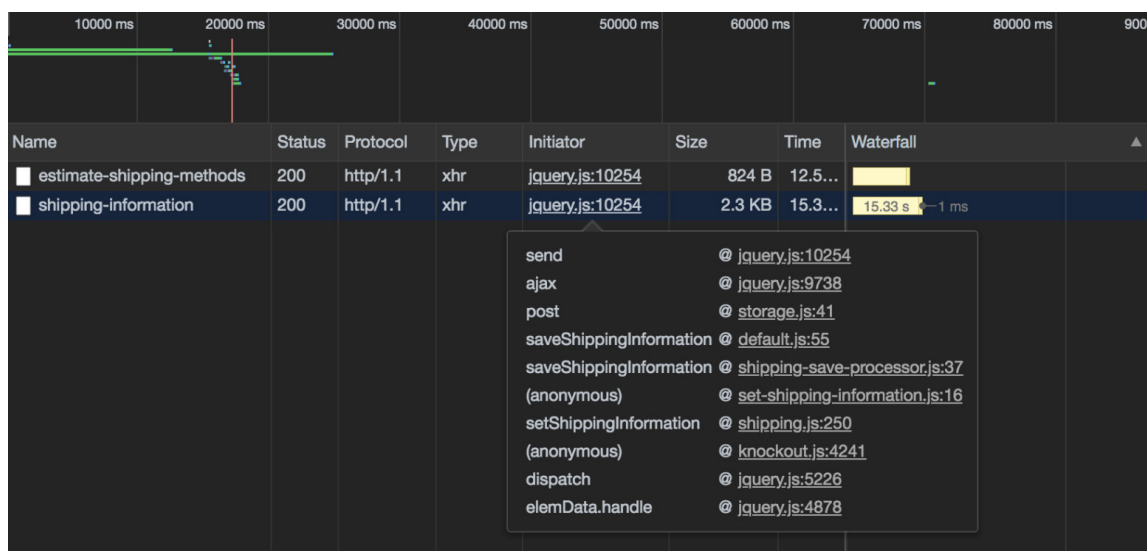
```
<item name="steps" xsi:type="array">
  <item name="displayArea" xsi:type="string">steps</item>
  <item name="children" xsi:type="array">
    <item name="shipping-step" xsi:type="array">
      <item name="sortOrder" xsi:type="string">1</item>
      <!-- ... -->
    </item>
    <item name="billing-step" xsi:type="array">
      <item name="sortOrder" xsi:type="string">2</item>
      <!-- ... -->
    </item>
  </item>
</item>
```

Debug the data flow of each step.

There is no substitute for you getting in there and stepping through the Javascript modules in the checkout.


There are multiple places to find the applicable module to start with, but one way is to start at the request and work backward. In the Chrome DevTools Network tab, one column is the Initiator (don't see it? right-click the column headers). Hover over the initiator to see a stack trace of the code that led up to the request. We triggered the following requests by simply completing the first step of the checkout:

Checkout



Payment Method:

☐ Check / Money order

☐  PayPal (Braintree)

☒ Credit Card (Braintree)

☒ My billing and shipping address

Jesse Maxwell

This provides a jumping-off point: `set-shipping-information.js`. The next call (`shipping-save-processor`) determines what processor to use for handling the address. Finally, extension attributes are added to the payload, and the appropriate URL is determined for sending the data, before it is sent to the Rest API.

How do you customize a step's logic?

This is a broad question. Depending on the logic you need to alter, you can either use a mixin for the Javascript component or XML (or a PHP Interceptor) for configuration. Read this [DevDocs article](#).

Customize the shipping step rendering and saving

How does Magento save information about the shipping address for different types of checkout (logged in with default address, without default address, not logged in)?

The persistence (POST request to the API) process is essentially the same across the various types with a utility looking up the applicable URL based on the quote (customer or guest). If the user is logged in, their addresses are part of the configuration embedded in the page (see `\Magento\Checkout\Model\DefaultConfigProvider::getCustomerData()` for the injection point). Each of them has a key which is used to communicate with the server which of those addresses was selected. If a new address was added, it is treated similar to the guest's address.

How does Magento obtain the list of available shipping methods?

It makes an API call to one of two endpoints: `estimate-shipping-methods` or `estimate-shipping-methods-by-address-id`. The first is expected to provide an address to base the lookup off of. The second sends an address key to use for getting the list of options.

Which events can trigger this process?

Any change to the shipping address.



HINT

It is possible to [trick it to update](#) as well.

How does Magento save a selected address and shipping method?

When the customer moves to the second step, an API call is made to save the `shipping-information`. The chosen carrier and address information is sent and saved into the customer's quote.

5.2 DEMONSTRATE UNDERSTANDING OF PAYMENTS

Describe the architecture of JavaScript payment modules

Add new payment method and payment methods renderers.

- Create a UI Component for the new payment method. This is a Javascript module that serves as a renderer and should extend `Magento_Checkout/js/view/payment/default.js`.
- Create a PHP `ConfigProvider` to obtain information from system config.
- Create another Javascript module to register the renderer.
- Create a `.html` template to display the new payment method.
- Integrate the new payment method's UI Component with the checkout by setting them to display in the `afterMethods` display area.



FURTHER RESOURCES:

For more information and a clear, step-by-step guide, refer to [this resource](#).

Modify an existing payment method.

Depending on the nature of the modification, it may be necessary to create a mixin. Some things can be done by hooking into existing architecture as shown in [this guide](#).

Demonstrate the ability to use the payment data flow for debugging and customizations

How does a payment method send its data to the server?

It is assembled through a number of Javascript modules and sent via a POST request to the server when the order is placed. There are a wide range of specific implementation aspects, but most of them provide some data to link the payment to the order (often a token).

What is the correct approach to deal with sensitive data?

It is important not to store sensitive data on your server. As a result, the best way to handle this payment data is to integrate with a provider such as Braintree. Payment tokens or nonces can be used to reference the payment type or transaction. Further, it is not safe to store sensitive info in localStorage.



RELATED READING

Check out [this article](#).

Demonstrate the ability to customize and debug the order placement process in JavaScript

Describe the data flow during order placement

The basis for this is: `Magento_Checkout/js/view/payment/default`. Its `placeOrder()` method initiates the process of transmitting the order information to the server. The `getData()` method is called and should return payment information. There is an `additional_data` key in the object that provides a good way to send additional payment information details to the server. The `Magento_Checkout/js/action/place-order` module takes that data, combines it and the cart ID and the billing address into an object and sends that to the server.

The `Magento_Checkout/js/view/payment/default` module is a good one to extend when you implement a custom payment method.

Braintree contains an example of integration. It extends the base `Magento_Checkout/js/view/payment/default` class. It has a `getData()` method that returns an object with the payment method code and the payment method nonce.

There are a number of methods that facilitate extension as well. For instance, after the order is placed, an `afterPlaceOrder()` method is called. It doesn't do anything in the core but is a good place to extend via a new module or mixin.

Which modules are involved?

We recommend reviewing these modules:

- Payment provider integration modules such as: `Magento_Braintree/web/js/view/payment/method-renderer/cc-form.js`

- `Magento_Checkout/js/view/payment/cc-form.js`
- `Magento_Checkout/js/view/payment/default.js`
- `Magento_Checkout/js/action/place-order`
- `Magento_Checkout/js/model/place-order.js`

How can the payment step be separated from the order placement?

The new step should have a higher sort order value than the payment step. It is then necessary to change the submit payment API call to use `set-payment-information`, instead of `payment-information`. The latter will submit the order, while `set-payment-information` saves it to be used later.



FURTHER RESOURCES:

DevDocs has a [topic](#) regarding adding a step.

Found a typo or error in this guide? We want to hear: joseph@swiftotter.com

EXAM NOTES:

Like Magento's other exams, there is no substitute for experience. We have provided you with a thorough overview of Magento's Javascript framework, but there were many concepts that we simply touched on without giving the detail that they may deserve. Magento recommends at least 1 year of experience, but due to the fact that many developers who work with Javascript do not solely develop

with it, it may take several years for the average front-end developer to reach the equivalent.

This exam had a number of questions that had specific nuances in the question that were important given the available answers. While that may seem normal for an exam, we encourage you to carefully consider all the details offered in each question and then choose the best question.

GLOSSARY

Terms:

- UI Component: an entire system comprised of XML, PHP, and Javascript. May be used to generally refer to one of those aspects.
- Javascript module: a Javascript file that contains a `define ()` function and returns a function - of which a jQuery widget would qualify.

All examples based on Magento 2.2 or higher.