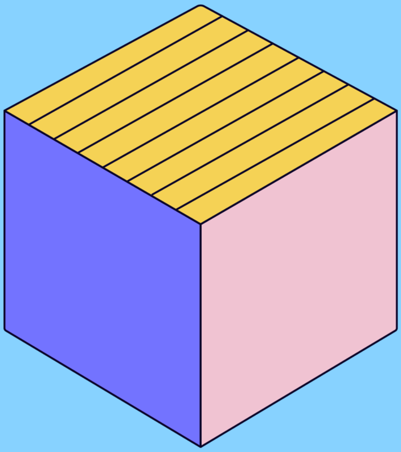


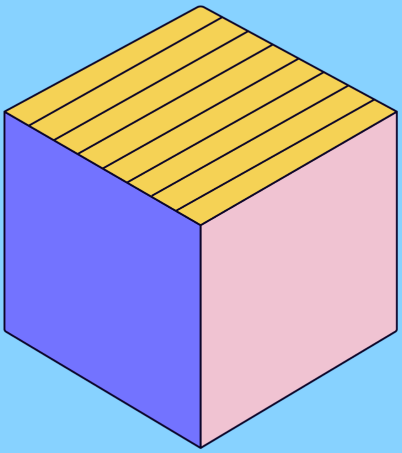
AD0-E720

Adobe Commerce Expert Frontend Developer

SwiftOtter

Joseph Maxwell and Vitaliy Golomoziy

Table of Contents



Introduction

Adobe Commerce Expert Frontend Developer, AD0-E720

SwiftOtter

Introduction

You have taken the first step toward becoming a Magento 2 Certified Expert Frontend Developer by downloading SwiftOtter's study guide. We've worked hard to produce a quality annotated eBook. Now it's your turn to get to work improving your Magento 2 knowledge and your future.

I do not recommend novices use this study guide, let alone take this test. You will not be able to pass by having little experience and then reading this study guide. Instead, start with the Professional Developer test, pass it and **then** go for this test.

If you are an expert, then this is the test for you (aptly named!). Review this study guide. Actively work to prove me wrong (then let me know!). Assume that I'm lying through my teeth to you on every single page. I'm being hyperbolic here as I'm giving you the very best information I possibly can. However, you *must* internalize every sentence written in this guide.

Magento is a world-class platform and highly skilled professional developers elevate the whole community. The better you are, the more everyone benefits.

The best way to pass the test is to know Magento 2.

- As you have questions, feel free to ask in our Slack channel - the 2nd biggest Magento-focused Slack community.
- The test is 50 questions and 100 minutes - 2 minutes per question.
- The test questions are scenario-based. You are provided information and a relevant question. You then choose the appropriate answer(s).
- There are three answers per question. Fewer answers does not make it easier.

All the best! Joseph Maxwell

We are SwiftOtter

We are focused, efficient, solution-oriented. We build sites on Magento and BigCommerce.

New sites, migrations and maintenance are our bread and butter. Our clients are our friends. Simple.

We hire the smartest people in the industry and pay them well. We provide this training first and foremost for our team, but also share this wealth with others, too.

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 - The database automation tool

How do you get the database from production to staging? Or back to local? And ensure that customer data is properly sanitized from the database? Or prevent those external API keys from trickling back to local and then trashing production data?

Meet Driver. This is a tool that allows you to automatically sanitize tables-with a snap-in for Magento.

Environments:

You can output to different environments (as in a different output for staging versus local).

Anonymization:

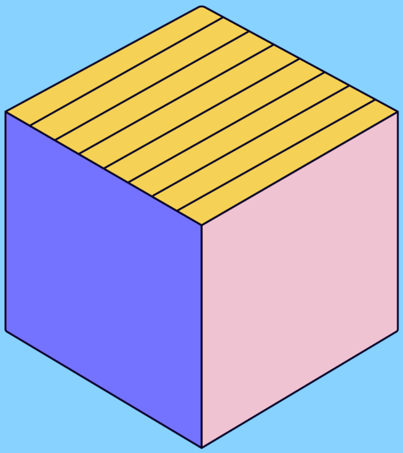
Ensure that customer data is properly cleared out. We have supplied a standard package for Magento 2 tables. You can easily create your own custom anonymizations.

How it works:

We *never* want to modify the local Magento database. Thus, we dump the database, push up to an RDS instance, run the transformations, export to a gzipped file and push to S3.

To load the data back from S3 into your staging or local environments, just run a command for this.

This tool has been transformational for SwiftOtter's processes.



Theme Management

Adobe Commerce Expert Frontend Developer, AD0-E720

SwiftOtter

1.01 Describe Adobe Commerce theme folder structure and how it relates to folder based themes

Points to remember:

- Local themes are stored in `app/design`.
- Composer-based themes can be stored anywhere.
  - As default unless a specific alternative is stated, Composer based modules/ themes will be located in the `vendor/` directory.
- `etc/`, `i18n/` and `web/` are directories found in most themes.

Local themes are stored in the `app/design` directory. If a theme is loaded through Composer, that theme can be located anywhere on the file system, but in most cases will use the default vendor/ directory within Magento.

See Objective 6.2 for more details regarding the Composer.

Magento uses the Composer autoloader. If you look at Luma theme's composer.json, you will see the `autoload.files` node, whose value is `registration.php`.

As the Magento application starts up, Composer executes each file as specified in the `autoload.files` section. `registration.php` then registers itself as a theme. The theme is now available.

Describe the different folders of a theme

`etc/`: this folder usually has one file: `view.xml`. `view.xml` provides configuration values for the theme in a structured format.

`i18n/`: this folder contains translations for the theme: `en_US.csv`.

`media/`: this folder usually has one file: `preview.jpg`. The preview image provides a sample of what the theme will look like when activated.

`web/`: the files and directories here will be eventually downloaded by website visitors. In one form or another, they will ultimately be accessible from `pub/static/`. LESS files will first be placed in `var/view_preprocessed` before being compiled and found in `pub/static`. As a rule of thumb, Magento recommends not to use this directory but rather place customizations of the theme into the appropriate directory within the module directory where the functionality originates (for example, checkout customizations should be placed under the `Magento_Checkout/web` directory).

`css/`: location of base Magento stylesheets. These will be exported to the `pub/static/[area]/[package]/[theme]/[locale]/css` directory.

`css/source/`: LESS files that implement styles for basic UI elements. Most of these styles are mixins for global elements from the Magento UI library. `theme.less` is also located here, which overrides values for the default variables.

`fonts/`: web fonts that will be utilized in the theme.

`images/`: images that are included in the theme. These are images that will not frequently change. For example, you would include an icon here but not a free shipping banner.

`js/`: theme-specific JS.

Module overrides:

When developing a theme, you will likely need to override another module's assets. These overrides reside in the theme folder, then the module's name. For example, in our `SwiftOtter_Flex` theme, we need to override `addtocart.phtml` in `Magento_Catalog`.

To accomplish this, we would copy `addtocart.phtml` into `app/design/SwiftOtter/Flex/Magento_Catalog/templates/product/view`.

The result is an easy-to-remember and replicable directory scheme.

Further reading:

- Theme Structure

Describe the different files of a theme

`composer.json` : this provides basic instructions, telling Composer information about the module. The most important information is the `autoload.files` node where Composer is informed about `registration.php`.

`registration.php` (required): this registers the module as a theme with Magento. It is important to note that the component name is made up as follows:

- `adminhtml` or `frontend`
- /
- Package name (often the developing company)
- /
- Theme name

Examples:

- `frontend/SwiftOtter/Flex`
- `frontend/Magento/luma`
- `adminhtml/Magento/backend`

`theme.xml` (required): this file describes the theme to Magento. You will see a `title` node, a `parent` node (optional), and a `media/preview_image` (optional) node.

Further reading:

- Theme Structure

##1.02 Demonstrate the ability to create a new theme (Inheritance/Fallbacks, design exceptions, theme properties)

Describe the relationship between themes

What type of relationships can exist between themes?

Parent/child.

In the theme's `theme.xml` file, you can specify the `<parent/>` node, like: `<parent>Magento/blank</parent>`. As such, a theme can be a part of (or the end of) many layers of other themes.

In the Magento world, the word "fallback" is used to describe the priority in which Magento loads the file. The reason why fallback is needed, is the fact that the "same" file may be located in multiple places; for example a phtml template can be located in the module's folder, theme's folder, and child theme's folder. So Magento has to choose one file out of the three possibilities. Working through this guide you will learn about fallback of:

- Phtml templates (Section 2)
- Layout updates (Section 2)
- Styles (Section 3)
- JavaScript modules (Section 4)
- Html templates (Section 4)

This hierarchy is what is used to determine the fallback sequence for theme inheritance (see `\Magento\Theme\Model\Theme.php` and `Magento/Theme/Model/Theme/`).

What is the difference between a parent theme and a child theme?

The only difference is that the child theme is the theme that is currently selected for display in the specified area. Any theme can be chosen for display (whether or not it specifies a parent in `theme.xml`).

Further reading:

- Theme Inheritance

How can the relationship between themes be defined and influenced?

The relationship is determined by the presence of the `<parent/>` node in `theme.xml`.

How is that taken into account when creating a custom theme or customizing an existing theme?

This information is useful to determine the fallback sequence. Fallback is used to determine a file to load in a sequence of other themes when the file does not exist in the current theme. The actual mechanism varies between file types (different for layout XML vs template files).

Theme properties

Theme configuration is located in two files: `theme.xml` and `etc/view.xml` We have briefly discussed `theme.xml`: it contains information about the parent theme, title, and preview image. `view.xml`, in turn includes an important configuration for different image types and their dimensions, variables, and js files to exclude (from bundling) See: DevDocs: Configure Theme Properties and `Magento/blank/etc/view.xml`

1.03 Demonstrate ability to extend existing themes

One may extend a theme in the following ways:

- Create a new theme that extends (inherits) another theme
- Extend theme variables and styles
- Modify JavaScript files and html templates
- Modify module's less, css, layout and template files

Create a new theme that extends (inherits) another theme

See previous section for details.

Extend theme variables and styles

There are two ways to customize styles - using fallback mechanism (see previous section), or using a special file (`web/css/source/_extend.less`) which is included automatically for every theme.

See more details here: Simple Ways to Customize a Theme's Styles

Modify JavaScript files and html templates

See section 4 of this study guide.

Modify module's less, css, layout and template files

See sections 2 and 3 of this study guide.

1.04 Demonstrate ability to customize transactional emails

Email templates in Magento are organized as follows:

- There is a file (`etc/email_templates.xml`) which describes the main information about the template.
- Email templates are technically html files with a custom syntax. Files are located in the `view/frontend/email` (or `view/adminhtml/email`) folder.
- In the template, one may specify different custom variables, use variables coming from the PHP code, or add custom styles and images.
- It is possible to customize email templates in the admin panel.

Xml configuration for email templates.

The file `etc/email_templates.xml` is defined in `magento-email/etc/email_templates.xsd` .

A great example of email templates is in the `Magento_Sales` module: Magento/Sales/etc/email_templates.xml.

Main configuration items which are specified in the xml file are: `id`, `label`, `file`, `type` (usually "html"), `module`, and `area`.

All options are self-explanatory.

Html template files for emails

Email templates are located in the `view/frontend/email` (or `view/adminhtml/email`) folders. See, for example: `Magento/Sales/view/frontend/email/order_new.html`.

In the html file, you can do the following:

- Include another html file by its id: `{{template config_path="design/email/header_template"}}`
- Use translation: `<h3>{{trans "Billing Info"}}</h3>`
- Use variables (usually coming from PHP code): `{{trans "%customer_name," customer_name=$order_data.customer_name}}`

Here you have a translation instruction applied to the string `"%customer_name"`. The `%` indicates that there is a variable (`customer_name`). The variable should be specified after the string: in this case: `customer_name=$order_data.customer_name`. Note that the `order_data` has been passed from the PHP code as an array with the `"customer_name"` key.

You can also apply some functions to the variables, such as: `raw` (leave the text as it is), `nl2br`, `escape` and so on.

Email styling will be covered in section 3.5 and the admin panel management in 5.5.

1.05 Demonstrate ability to apply translations

Create and change translations

Demonstrate an understanding of internationalization (i18n) in Magento. What is the role of the theme translation dictionary, language packs, and database translations?

Internationalization has been a core Magento feature since its early days. Magento 2 maintains strong support across the entire platform.

Magento includes a feature to locate all translatable strings within a particular path. You can utilize this for an entire Magento installation or just for a module or a theme.

To find all translatable strings for a module (or modules):

```
bin/magento i18n:collect-phrases app/code/SwiftOtter/Test
```

To assist in building a language package, you need to locate all strings within the Magento application. You can run this command to obtain this information:

```
bin/magento i18n:collect-phrases -m
```

When run with the `-m` flag, two additional columns are added: `type` and `module`. `type` is either `theme` or `module`. The `module` column represents the module that utilizes this translation.

Theme translation dictionary

A theme translation dictionary allows you to specify translations for words used in a theme or a module. These phrases are placed in a `.csv` inside your module or theme's `i18n` directory (`app/code/SwiftOtter/Test/i18n/de_DE.csv`).

Translations specified for the theme or module are the first two sources of translation data. These translations can be overridden by a language package or in the database.

Further reading:

- Generate a Translation Dictionary
- Use Translation Dictionary to Customize Strings

Language packs

A language pack allows you to translate words used anywhere in Magento. The source for this is the `bin/magento i18n:collect-phrases` command with the `-m` flag. This searches the entire Magento application (including modules and themes) for all translatable strings.

The output from this command is the fundamental ingredient to a language pack.

Once you have translated the strings, you then execute this command (substituting the path to the CSV file and specifying the target language):

```
bin/magento i18n:pack /absolute/path/to/file.csv de_DE
```

You must then create a new language module within the `app/i18n/SwiftOtter_FR/` (or something similar). This contains the usual module files (`registration.php` uses the `\Magento\Framework\Component\ComponentRegistrar::LANGUAGE` component type). It also includes a `language.xml` file:

```xml
<?xml version="1.0"?>
<language xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
        xsi:noNamespaceSchemaLocation="...">
  <code>fr_FR</code>
  <vendor>SwiftOtter</vendor>
  <package>fr_fr</package>
  <use vendor="Magento" package="fr_FR"/>
</language>
```

You can include multiple `<use/>` nodes in the `language.xml` file. If Magento does not find a string in the included language package, it will search through each `<use/>` (and subsequently those language package's `<use/>` nodes) until it finds an applicable translation.

If no translation is found, the module or theme's own translation is used. If none is found, the original, untranslated text is returned.

Inside the language package, include the CSV generated above named as the contents of the `<code/>` node with a `.csv` suffix.

Further reading:

- Create a Language Package
- German Locale Pack
- French Locale Pack

Database translations

Database translations are the easiest to implement, but the most difficult to transfer from installation to installation. They are found in the `translation` table.

To create a new translation, the easiest is to turn on Inline Translation (Store > Configuration > Developer > Translate Inline). You could also insert new rows in the `translation` table manually with the limitation being the need to determine the module to associate the translation to.

While translating with Inline Translation, disable the `Translations`, `Block HTML`, and `Full Page` caches.

Understand the pros and cons of applying translations via the `translate.csv` file versus the `core_translate` table. In what priority are translations applied?

Any translations found in the `translate.csv` are easily replicated to other instances where this file is utilized. However updating a file is usually less convenient than using the translate inline feature for the Magento frontend.

Translations are applied in this order:

1. Module translations
2. Theme translations
3. Translation package
4. Database translations

As a result, database translation is good for overriding other translations where necessary.

Translate theme strings for .phtml, emails, UI components, .js files

PHTML:

```
<?= __('Shopping cart');?>
<?= __("There are %s items in your cart", $count); ?>`
```

Email templates:

```
{{trans "Shopping Cart"}}
{{trans "%items are shipping today." items=shipment.getItemCount}}
```

JavaScript translation:

See section 4.5 for details.

Further reading:

- DevDocs: Translate Theme Strings
- Mage2.tv: Localization in JavaScript

1.06 Given a set of use cases, identify when to place files in app/code or app/design

`app/code`

This folder is used for custom modules, so everything that falls in the custom module should go there. Usually there is a `view/frontend/web/css/source/_module.less` file which is a starting point for module-related styles. Html, phtml, js, and layout files that are related to a module should be placed into a module's folder as well.

`app/design`

This folder is used for custom themes. So everything that is theme-specific should be placed here. For example, general theme variables and common styles are always theme-specific.

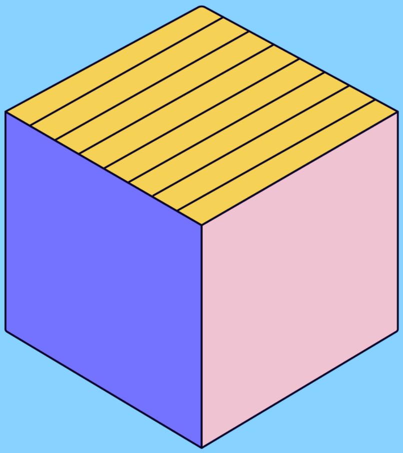
It may be a bit challenging, sometimes, to identify what exactly goes to the module and what to the theme. The rule of thumb is the following: anything that is more or less theme independent should be placed in the module's folder, and anything that is theme-specific in the theme's folder.

The second aspect is customizations. Sometimes, assuming there is a custom theme, one uses the theme to override a file (a phtml template for example), just because it is simpler. Generally speaking, this is not best practice. If you need to override a phtml template for a given module, it is better to do it in the custom module, using a layout update.

Another side of the same problem is a situation when there is no custom theme, but some general styles have to be changed. In this case, changes may be placed in a custom module. However, this is not best practice either.

Finally, to see an example of the theme-specific versus module-specific files, see the Magento_Catalog module:

- `Magento/Catalog/view/frontend/web/`
- `app/design/frontend/Magento/luma/Magento_Catalog/`



Layout XML & Templates (phtml templates)

Adobe Commerce Expert Frontend Developer, AD0-E720

SwiftOtter

2.01 Demonstrate the ability to utilize layout XML instructions

Demonstrate knowledge of all layout XML directives and their arguments

What layout XML elements exist and what is their purpose?

The basic building blocks of layout XML elements are `<block/>` and `<container/>`. Most other layout instructions make modifications to blocks and containers.

Containers

Containers contain or group blocks and other containers. If you look in the root layout XML file, you will see the top-most element in `Magento/Theme/view/base/page_layout/empty.xml` is a container.

Containers are helpful in providing a more extensible means of grouping and rendering child blocks because the HTML tag and class can be specified in XML. Using blocks with an iterator, or a `ListText` block does not have this advantage. However, the `ListText` block has the advantage that data values can be set via the layout XML `<arguments>` node. Containers don't allow for `<arguments>`.

Notable container attributes (Layout Instructions: Container)

- `name`: required. This defines how you will reference this container from other areas in the system.
- `htmlTag`: a tag to wrap the child block output. Required if htmlClass is specified.
- `htmlClass`: the class(es) to apply to the wrapper HTML tag.

ReferenceContainer

The extensible aspect of a container is to be able to reference it throughout layout XML. This capability is leveraged with the `<referenceContainer/>` tag.

Here's an example from `Magento/Catalog/view/frontend/layout/catalog_product_view_type_simple.xml`:

```
<referenceContainer name="product.info.type">
    <block name="additional.product.details"
      template="SwiftOtter::product.details.phtml">
        <arguments>
            <argument name="viewModel" type="xsi:object">
              SwiftOtter\Flex\Block\ProductDetails
            </argument>
        </arguments>
    </block>
</referenceContainer>
```

The above example will include a new logic class (using the recommended Template Block + ViewModel approach) to render additional product details. The template from this example is automatically rendered to the page within the container `product.info.type`.

Blocks

Blocks are a foundational building unit for layout in Magento. They are the link between a PHP block class, which contains logic and a template which renders content. Blocks can have children and grandchildren (and so on). Information can be passed from layout XML into the block via the `<arguments/>` child node. For those of a hybrid discipline (backend developers) this could include view models (View Models in Magento 1 and 2) or some static values to be utilized in the template.

Notable block attributes (Layout Instructions: Block)

- `class`: defaults to `\Magento\Framework\View\Element\Template` and does not need to be specified. In fact, Magento recommends against specifying a class (see note in the header comments for the Template class in case the block only is required to output a template).

- `name`: used to interact with the block from other locations in the frontend.
- `before` / `after`: places the block before or after another element, as identified by its name. Or, you can specify a `-` (dash) to place the block at the beginning or end respectively.
- `template`: the path to the template. You should always use Magento module path notation. Example: `SwiftOtter_Hero::hero.phtml`. If the module name is omitted, Magento will attempt to determine it based on the block name, which fails if a generic block class is used or the block class is changed with layout or di XML (more details: Magento 2 Template Paths).
- `cacheable` (default `true`): if specified as `false`, the entire page will not be cached with Full Page Cache. Because this negatively impacts performance, it is best to never add to blocks that are on pages where the caching is applied. Instead, follow the directions in this Magento Stack Exchange post on how to utilize Magento's ESI system if the block content is public, that is, the identical for all visitors, or use customer-data sections in case the content is private (that is, specific for each visitor). More information on including private data on cached pages can be found on Mage2.tv: Full Page Cache.

Further reading:

- Layout XML Instructions

ReferenceBlock

Like containers, referencing another block allows you to affect the output of another block.

- `<referenceBlock name="product.info.sku" remove="true"/>`: completely removes the block from layout.
- `<referenceBlock name="product.info.sku" display="false"/>`: prevents a block from displaying (the associated PHP classes are still loaded).
- Change the template of an existing block:

```xml
<referenceBlock name="product.info.sku">
    <arguments>
        <argument name="template" xsi:type="string">
            SwiftOtter_Flex::new-sku.phtml
        </argument>
    </arguments>
</referenceBlock>
```

- Note that changing the template here will break fallback patterns that depend on the original file chosen. As such, this might be problematic for module developers.
- For example, let's say you are developing a module and want to redirect the SKU's template to a custom one in your module. A merchant, who uses your module, installs a custom theme. When they browse to the product page, the SKU might not look correct because the theme they installed did not take into account the updated SKU template.

What is the purpose of the attributes that are available on blocks and other elements?

The idea is to place configuration into one location (alongside the other configuration that is controlling the look of the page). Layout XML configures containers and blocks. It would be easier to maintain extra details for containers and blocks when they are not buried in a mass of HTML. In addition, this opens up the possibility for module developers to easily adjust these parameters without having to override many templates.

Other important layout instructions:

- `<update handle="customer_account"/>`: this includes instructions from another layout handle. Example: `Magento/Sales/view/frontend/layout/sales_order_view.xml`
- `<move element="existing.element.name" destination="target.element"/>`: this moves an element (block or container) into another element (block or container).

   - The `move` element also contains `before` and `after` attributes so you can change the element's placement within a parent if so desired.

2.02 Demonstrate the ability to create new page layouts

Describe page layouts and their inheritance

How can the page layout be specified?

The page layout is specified in a layout XML file, in the root `<page/>` node, like: `<page layout="2columns-left" …></page>`

What is the purpose of page layouts?

The purpose is to create a structured and common set of layout instructions to render pages. Most pages on a website can be categorized as fitting into a 1 column, 2 column, or 3 column container system. These page layouts can be selected throughout the admin panel to provide a specific layout by page.

How can a custom page layout be created?

To create a custom layout, you must first define the layout, define the common layout instructions, and then utilize that layout (as seen in the first answer to this section). In our example below, we will create a text layout. This is similar to the two column layout example except the text area is narrower.

Page layouts only contain containers. This is different from the page configuration (XML files with a layout handle as their file name, and stored in `view/[area]/layout`) which can contain blocks and containers.

The definition of the layouts are stored in your module's `layouts.xml` file, like `app/code/SwiftOtter/Test/view/layouts.xml`. See the following example.

Defining the layout:

In a module's `view/[area]` directory, create a `layouts.xml` file:

```
<!-- app/code/SwiftOtter/Test/view/layouts.xml -->
<?xml version="1.0" encoding="UTF-8"?>
<page_layouts xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
        xsi:noNamespaceSchemaLocation="urn:magento:framework:View/PageLayout/etc/layou
    <layout id="text">
        <label translate="true">Text layout</label>
    </layout>
</page_layouts>
```

Defining the common layout instructions:

```
<!-- app/code/SwiftOtter/Module/view/page_layout/text.xml -->
<?xml version="1.0"?>
<layout xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xsi:noNamespaceSchemaLoc
    <!-- notice foundation page layout of 2columns-right: -->
    <update handle="2columns-right"/>

    <!-- Instead of adding to body tag which isn't possible from page_layout xml -->
    <referenceContainer name="page.wrapper" htmlClass="page-wrapper page-layout-2colum
</layout>
```

Where are the existing page layouts used?

Page layouts can be customized in the admin panel on products, categories, or CMS pages.

How can the root page layout be specified for all pages and for specific pages?

Set the `layout` parameter on the `<page />` node to the ID of a layout specified in one of the `page_layout` XML files. The node can otherwise be blank if desired. Specific pages can be targeted using the layout XML's filename.

Further reading:

- Layout File Types

2.03 Understand the difference between extending/merging and overriding XML

Describe layout XML override technique

How can layout XML be overridden?

It is not ideal to override layout. However, as described by this article (Override a Layout), there are some situations where overriding XML layout is inevitable.

For example, to completely erase the existing Magento product page layout in our SwiftOtter Flex theme, you would create a new XML file in: `app/design/SwiftOtter/Flex/Magento_Catalog/layout/override/frontend/catalog_product_view.xml`.

Note that these files are placed into the `override` folder instead of directly inside the `layout` folder. XML files that are directly placed in the `layout` directory are merged, while XML files that are placed in the appropriate subdirectory of `override` replace the original file.

All layout instructions from the Catalog module's original `catalog_product_view.xml` will be eliminated.

How can layout overriding be used in theming?

If an existing layout file contains an instruction that you cannot change by extending, then overriding that layout file might be the only recourse. In our experience with building Magento websites, we have yet to come across a time where this is necessary.

What are the consequences of layout overrides during upgrades?

New blocks or containers could be added to the core layout file with a new version. If that file was overridden, other modules that depend on specific elements would not have access to them (and their functionality would be halted) unless you manually added them to your theme's override.

What is the effect of layout overrides on compatibility?

Overriding layout files circumvent any changes in core files and increase chances of trouble during upgrades. As such, it must be viewed as the ultimate last result.

Understand layout merging

What is layout merging?

Layout merging is the process of assembling all of the layout XML files into one large XML document (`\Magento\Framework\View\Model\Layout\Merge::loadFileLayoutUpdatesXml()`). When rendering the page, Magento locates the layout instructions for the particular handles (`\Magento\Framework\View\Model\Layout\Merge::fetchPackageLayoutUpdates`) and merges them together. Magento then traverses this document instantiating PHP classes for each block and then rendering the output.

Layout is merged in order that the modules are loaded (Component Load Order). To quickly see the loading order of modules, check the `app/etc/config.php` file. If two conflicting instructions are given, the module that loads last wins.

When you are building a new theme, you will need to make adjustments to the layout XML. While you can override layout XML files (see discussion above), this is less than ideal for upgradeability purposes as any updates that Magento ships will not be present in the overridden files.

Instead, utilize the layout XML merging directories. For example: `app/design/SwiftOtter/Flex/Magento_Catalog/layout`. We can create layout XML files that will be merged with layout XML files in the `Magento_Catalog` module. As such, we can use instructions like `<move element="element.to.move" destination="can.be.same.parent" after="element.to.be.placed.after"/>`

If you want to see the results of the layout merging on your developer machine, do the following:

- Navigate to `\Magento\Framework\View\Result\Page`
- Locate the `render()` method.
- After the `$output = $this->renderPage();` method, add:
    - `$output .= $this->getLayout()->getXmlString();`

Go to a page on the website and view source. The XML will be appended after the HTML tag.

How do design areas influence merging?

Layout XML instructions are first merged in the `base` area and then by the area that applies to the current request (`frontend` or `adminhtml`).

Here is the order in which layout XML is merged:

- Module base files loaded
- Module area files loaded
- Sorted according to their module priority (array index of module's position in `app/etc/config.php`)
    - If their priorities are equal, they are sorted according to their alphabetical priority.
- Theme files (from furthest ancestor to current theme)
    - Layout files loaded
    - Override files replaced
    - Theme override files replaced

Source: `vendor/magento/framework/View/File/Collector/Base.php`

How can merging remove elements added earlier?

The only way to remove elements added earlier is through the `remove` attribute on the `referenceBlock` or `referenceContainer` tags:

```
<referenceBlock name="info.product.sku" remove="true"/>
```

What are additive changes and what are overriding changes during layout merging?

Additive changes are when you create new XML elements that are in addition to what has already been written. These changes will likely affect existing elements (setting new arguments, for example).

Let's look at an example from `Magento_Catalog/view/frontend/layout/catalog_product_view.xml`. Locate the `product.attributes` block in this file:

```
<block class="Magento\Catalog\Block\Product\View\Attributes" name="product.attributes"
  <arguments>
    <argument translate="true" name="title" xsi:type="string">More Information</argume
  </arguments>
</block>
```

Let's create our own, in `SwiftOtter/Flex/Magento_Catalog/layout/catalog_product_view.xml`:

```
<?xml version="1.0"?>
<page xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xsi:noNamespaceSchemaLocat
    <body>
        <referenceBlock name="product.attributes">
            <arguments>
                <argument translate="true" name="test" xsi:type="string">Test Argument
            </arguments>
        </referenceBlock>
    </body>
```

```
</page>
```

When setting a breakpoint in `vendor/magento/module-catalog/view/frontend/templates/product/view/attributes.phtml`, you will see that the `$block->_data` array has a new key, `test`, with the value of `Test Argument`. This is considered an additive change.

You can change the argument name to `title` and notice the difference in the `$block->_data` variable. As such, you are overriding the existing value for `title` and are specifying your own. This is considered an overriding change.

Understand processing order of layout handles and other directives

In what order are layout handles processed?

They are processed in the order in which the handles were added to the Merge class `\Magento\Framework\View\Model\Layout\Merge`. The `default` handle is loaded first. Then, in order of when they were added through the request lifecycle.

In the above class, see the `load()` method. In this method, the layout is merged as it loops through the associated layout handles.

As a side note: this can cause some trouble if you are relying on a custom-injected layout handle to adjust the design on a page. As such, there may be instances where you need to go to extra lengths to inject a layout handle earlier in the application's lifecycle.

In what order is layout XML merged within the same handle?

This would then be according to the order in which the layout files are loaded. See the in-depth topic on Section 3.6.

How can the processing order be influenced?

The processing order and merging of layout XML is determined by the module load order. Adjusting the `<sequence/>` setting for modules will influence the processing order of layout

XML.

For backend developers, you can use plugins to inject your layout handles earlier in the process. For example, if you want to add a layout handle before the `catalog_product_view` one, you need to create a `before` plugin for `\Magento\Catalog\Helper\Product\View::initProductLayout()`.

What are common problems arising from the merge order of layout declarations?

The most common problems are:

- Accidental overrides: where you specify `<block name="...">` which happens to be the same name as another block.
- Wishful overrides: where you want to override another block, but you specify a different name and the block is injected twice.

2.04 Demonstrate how to pass and utilize arguments to templates

Points to remember:

- Templates are phtml files; they have no arguments.
- Phtml templates may only be rendered via the layout mechanism in connection to the `Magento\Framework\View\Element\Template` block.
- Blocks allow arguments passed through the layout.xml, di.xml and directly in the php code. In this section, we are only interested in the first option.
- Sometimes it's useful to encapsulate all variables needed for a template in a separate php class (called ViewModel).

Render values of arguments set via layout XML

How can values set on a block in layout XML be accessed and rendered in a template?

In this example customization:

```
<referenceBlock name="product.info.sku">
    <arguments>
        <argument name="test_value" xsi:type="string">11111</argument>
    </arguments>
</referenceBlock>
```

The value for `test_value` will be available in the template like: `$block->getData('test_value')` (or `$block->getTestValue()`).

Further reading:

- Adding Frontend Assets

Usually when developing a new phtml template associated with a block, we have a need to pass variables to the template. Above we've seen that we can do it via `<arguments>` layout instruction, which is quite a powerful tool, however sometimes (almost always, actually) we need more dynamics and flexibility. In order to achieve it, we create view models. `ViewModel` (in this context, don't mess it up with knockout ViewModels!) is a PHP class which provides variables to a template. From within a template, you can access a view model in the same way as described above, and then fetch all variables from the view model itself.

Further reading:

- DevDocs: View Models

2.05 Demonstrate ability to create and customize templates

First, we'll discuss creating new phtml templates (and associated blocks), and later we'll cover how to override templates for existing blocks.

In order to create a new, functional phtml template one has to take the following steps:

- Create/modify a layout update file and add a new block.
- The easiest case is using a block of the type `Magento\Framework\View\Element\Template`.
- One may create a custom block extending `Magento\Framework\View\Element\Template` but this is not necessary.
- Create a phtml file itself, in the module's folder `view/frontend/templates/<path-to-phtml-file>.phtml`. Obviously adminhtml templates go to the `view/adminhtml/templates` folder. Pay attention to the folder name - `templates` not `template`!
- A template can also be created in the theme's folder, however it is better to place new templates in the module, unless there is a reason to have it in the theme.
- Take action in order for a block to be rendered. These could be assigning a block as a child of a container or assigning a block as a child of another block, in which case one usually has to override the parent's block template and render the newly created block manually using `<?php $this->getChildHtml(".."); ?>` instruction.

Since template files may have the same name (for example, different modules may have different `price.phtml` files, which is different from the catalog's one), we need an unambiguous way to reference a phtml file in the layout. For doing so, we add the file name (relative to the templates folder) after the module name. So the file `SwiftOtter/Module/view/frontend/templates/some/custom/price.phtml` will be referenced as: `SwiftOtter_Module::some/custom/price.phtml`

Assign a customized template file using layout XML

How can a customized template file be assigned to a block using layout XML?

```
<block name="custom.block" template="SwiftOtter_Module::test.phtml"/>
```

The provided template path can be broken down into two arguments, which are located on either side of the scope separator `::`.

The first states the area to look in, in this case `SwiftOtter_Module`, which looks in the corresponding module:

```
app/code/SwiftOtter/Module
```

The second part `test.phtml` is the path within this module's template folder (`view/[area]/template`): in this case that completes the path:

```
app/code/SwiftOtter/Module/view/frontend/template/test.phtml
```

If you would like to change the template and use another one:

```
<referenceBlock name="custom.block" template="SwiftOtter_Module::product-attribute.phtm
```

How does overriding a template affect upgradability?

When overriding a template file by assigning a new template via layout XML, you effectively break the fallback path for a template. This prevents themes from modifying the template.

What precautions can be taken to ease future upgrades when customizing templates?

- Backend developers: consider a plugin for the block (this would prevent using view models in such a case) that would only turn on the customized template when necessary.
- Avoid overrides where possible.

Override a native template file with a customized template file, using the design fallback.

How can the design fallback be used to render customized templates?

You can customize a template by adding it to your custom theme. For example, to modify `vendor/magento/module-catalog/view/frontend/templates/product/list.phtml`, use the following path:

```
app/design/frontend/SwiftOtter/Flow/Magento_Catalog/view/frontend/templates/product/lis
```

Further reading:

- [Templates Basic Concepts]()

How does that influence upgradability?

While the chances are usually slim, there can be broken functionality that occurs when upgrading. The reason is that the template that was overridden may have been enhanced along with other aspects of the system that depended on the template. If the overridden template is not upgraded, things can break.

The most notorious example was from Magento 1 days. SUPEE-9767 enabled form key checking on the checkout. With the patch, Magento added the form key to the templates. But many checkouts had at least some customized template files which then did not include the form key. While this was easy to catch with testing, it added to the overall implementation time for this upgrade.

How can you determine which template a block renders?

The easiest way is to enable template hints, which can be activated using the command-line: `bin/magento dev:template-hints:enable`. Or in the admin area, they are in the following locations:

Frontend: Stores > Settings > Configuration > Advanced > Developer > Debug > Enabled Template Path Hints for Storefront > Yes

Admin: Stores > Settings > Configuration > Advanced > Developer > Debug > Enabled Template Path Hints for Admin > Yes

The next step is to look through the Magento code. The template is often set in layout XML, so this is the best place to start. Locate the block that renders the template in XML and see if the template is set. If so, it will contain the reference to the template. In some places, though, the template is defined in the PHP class itself. If this is the case, open the block class and look for the `$_template` property or some place where it is set.

Further reading:

- [StackExchange: Template Path Hints]()

Describe conventions used in template files

What conventions are used in PHP templates?

- `$this` no longer applies to the rendering block. Use `$block` or `$block->getData('view_model')` to obtain access to the instigating object or its data.
- Always type hint variables that are automatically imported (`$block` , `$viewModel`).
- Never use squiggly braces: this is a code smell that indicates your block or view model should be doing more work.
- If you need to use a loop, use the `foreach` > `endforeach` constructs.
- Keep templates to a reasonable minimum. Massive 500 line files are a code smell.
- Magento now uses `<?=` instead of `<?php echo` .
- Always translate strings in templates that will be displayed to user.

Why aren't the common PHP loop and block constructs used?

Magento templates do not use the `if(){ /* … */ }` constructs because squiggly braces are harder to discern in HTML.

Further reading:

- StackOverflow: Difference between `if () { }` and `if () : endif;`

Which common methods are available on the `$block` variable?

- `getRootDirectory()`
- `getMediaDirectory()`
- `getUrl()`
- `getBaseUrl()`
- `getChildBlock($alias)`
- `getChildHtml($alias, $useCache = true)`
- `getChildChildHtml($alias, $childChildAlias = '', $useCache = true)`: this method returns the HTML from a grandchild block.
- `getChildData($alias, $key = '')`: calls `getData` on a child block.
- `formatDate($date = null, $format = \IntlDateFormatter::SHORT, $showTime = false, $timezone = null)`
- `formatTime($time = null, $format = \IntlDateFormatter::SHORT, $showDate = false)`
- `getModuleName()`
- `escapeHtml($data, $allowedTags = null)`
- `escapeJs($string)`
- `escapeHtmlAttr($string, $escapeSingleQuote = true)`
- `escapeCss($string)`
- `stripTags($data, $allowableTags = null, $allowHtmlEntries = false)`
- `escapeUrl($string)`
- `getVar($name, $module = null)`: locates a value from the themes `etc/view.xml`

How can a child block be rendered?

```
$block->getChildHtml('child-name');
```

How can all child blocks be rendered?

`$block->getChildHtml();`

How can a group of child blocks be rendered?

Groups are a little-known part of the Magento layout system. These methods are found in `vendor/magento/framework/View/LayoutInterface.php`. The primary example of groups in the Magento core is on the product detail page's tabs and can be seen rendered here: `Magento/Catalog/view/frontend/templates/product/view/details.phtml`. Rendering them involves obtaining all of their names using `getGroupChildNames` and then rendering each block by name in a loop.

2.06 Apply template security (escaping output)

Demonstrate ability to escape content rendered and template files

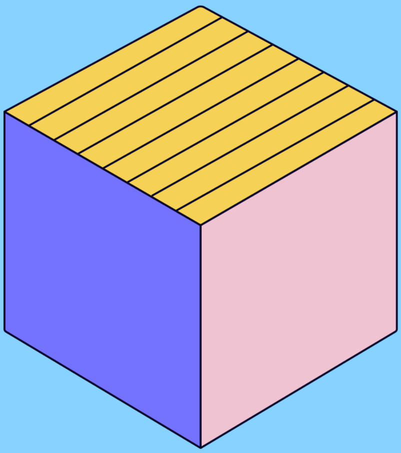
How can dynamic values be rendered securely in HTML, HTML attributes, JavaScript, and in URLs?

This is possible by utilizing the block's built-in methods.

- HTML: `$block->escapeHtml('value', $allowedTags);`
- HTML attributes: `$block->escapeHtmlAttr('value', $escapeSingleQuote);`
- JavaScript: `$block->escapeJs('value');`
- URLs: `$block->escapeUrl($url);`

Further reading:

- XSS Prevention Strategies



Styles

Adobe Commerce Expert Frontend Developer, AD0-E720

SwiftOtter

3.01 Identify the purpose of styles-m.less, styles-l.less, print.less

Magento utilizes the LESS preprocessor to simplify theming. It's goal is to keep styles more concise by providing variables, handling nested selectors, and allowing common functions (mixins). Browsers cannot interpret LESS so it must be compiled into CSS.

The most common LESS files that are compiled into CSS are: `styles-m.css`, `styles-l.css` and `print.css`. These files are included in [`Magento_Theme/layout/default_head_blocks.xml`](). According to a convention, all LESS files that are directly included and compiled to CSS do not start with an underscore.

- `styles-m.css`: mobile-friendly styles. In trying to reduce the download size for websites, `styles-m.css` contains any styles necessary for viewing on mobile.
- `styles-l.css`: additions for desktop styles. This contains the additional styles for displaying on a screen that is 768px or wider.
- `print.css`: additions for print styles.

There are other files that do not begin with an underscore and are individually output: (for example: [`Magento/blank/web/css/*`]()).

To insert an external stylesheet into the `<head/>` tag, you can use the `<css/>`, or the `<link/>` tag in layout XML:

```
<page xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xsi:noNamespaceSchemaLocat
    <head>
        <css src="css/local-stylesheet.css"/>
        <css src="https://swiftotter.com/external.stylesheet.css" src_type="url" />
        <link rel="stylesheet" type="text/css"
                src="https://swiftotter.com/another.external.stylesheet.css" src_type=
    </head>
```

```
</page>
```

The `<link/>` can also be used to add JavaScript resources. To make sure your code is clear, we would recommend adding CSS resources using the `<css/>` element.

Further reading:

- [DevDocs: Include Static Resources]()

3.02 Describe the difference between specific partial files (_extends.less, _extend.less, _theme.less, _widgets.less, _module.less)

Describe the purpose of `_module.less`, `_extend.less`, `_extends.less`

Demonstrate LESS has no fallback capabilities and therefore Magento created @magento_import directives to enable FE devs to inject or replace parts of existing less structures of modules and themes.

LESS does not have the ability to dynamically find and include partials. As a result, every path must be included individually in the primary stylesheet. To work around this limitation, Magento provides a `//@magento_import` directive. This will search for files through many locations and include each version that it finds. Note the use of the `//` as a comment to prevent any errors with LESS. Using the `//@magento_import` directive allows the inclusion of a specific file from each module with just one line.

Reference: [`Magento/blank/web/css/styles-l.less`]()

Here are some of the primary uses of `//@magento_import`:

`_module.less`: This is the main entry point of a module's stylesheet. For a module, it can have partials of its own, if desired, and should place them in a `/module` directory next to this file. For a theme, this would override the core module's primary stylesheet. As a result, this would be done in cases where a significant portion of the styles are being updated.

`_extend.less`: This should mostly or entirely be empty in modules. For themes, if most of the core module's stylesheets are good and the goal is only to update a few things, the easiest way to do that is to create an `_extend.less` file. This file is loaded after `_module.less` and consequently will apply given the same selector.

`_extends.less` (`Magento/blank/web/css/source/_extends.less`): this file contains a massive list of abstract selectors that can be "extended" from mixins.

`_theme.less` is a basic file that contains theme variables. Usually you do not have to override this file, unless you want to override all variables from the parent theme. See `Magento/luma/web/css/source/_theme.less` for more details.

`_widgets.less` a file that is supposed to include styles for widgets, and it is included automatically. However, native themes do not have this file.

3.03 Demonstrate the ability to override or extend Magento LESS

How to override a LESS theme file:

We need to modify the availability notice on the product page. The merchant has requested that this text is very large and prominent on the page. You have already created the SwiftOtter/Flex theme. How do you update the font size for this element?

Since this is a small change, the best might be to leverage `_extend.less` by creating one in `app/design/SwiftOtter/Flex/Magento_CatalogInventory/web/css/source/_extend.less`.

However, if this is the first of a litany of changes, override the LESS file.

To do this, copy `vendor/magento/theme-frontend-luma/Magento_Catalog/web/css/source/_module.less` into `app/design/SwiftOtter/Flow/Magento_Catalog/web/css/source/_module.less`. Modify the necessary code. Clear `var/view_preprocessed` and `pub/static` and refresh the page.

There are no ways to "extend" a less file per se (i.e. create another less file that will merge with the original one). However one may "extend Magento LESS" in a way of adding new less files, override variables, and so on. This functionality is described in Section 3.2 of the current guide.

3.04 Explain the core concepts of LESS

Explain core concepts of LESS

Describe features like file import via `@import` directive, reusable code sections via mixins together with parameters and the usage of variables. Demonstrate your understanding of the special variable `@arguments`.

`@import` directive:

While Magento contains hundreds of `.less` files, it compiles those files back into a number of `.css` files. LESS provides an `@import` construct to import a file. All files can import other files. Eventually an entire tree is built of imported files. A LESS convention is that all files that are included by the @import construct starts with an underscore in its filename.

The `styles-l.less` file contains this line:

```
@import '_styles.less';
```

This is including `_styles.less`, which includes other files.

Further reading:

- DevDocs: The `@import` directive rules of usage

- Less Css: @import At-Rules

Mixins

This is the equivalent to a method or a function in another programming language. Mixins return CSS instructions. The goal is that you write it once in a mixin and then reuse that throughout the application's theme (DRY principle).

The way it works is you write some LESS instructions (which, in this case, are verbatim CSS):

```
.element-selector {
  background-color: #ff0000;
}
```

And then call it:

```
.another-element-selector {
  .element-selector();
}
```

This results in:

```
.another-element-selector {
  background-color: #ff0000;
}
```

Mixins can also contain parameters. Using our above fictitious example:

```
.element-selector(@color) {
  background-color: @color;
}
```

This is called:

```
.another-element-selector {
  .element-selector(#0000ff);
}
```

Mixin parameters can be supplied in a format similar to a JavaScript object and many in Magento are set up this way. When this is the case, one can pass in a subset of arguments by including the parameter's name along with the argument value.

Further reading:

- Mixins

Variables:

Variables allow you to specify a value in one place and then use it in other locations (just like with programming). Magento makes heavy use of LESS variables (hopefully, someday we will see them simplified).

Further reading:

- Variables

@arguments

`@arguments` takes the arguments that were passed into a mixin and renders them in that order.

Further reading:

- The `@arguments` Variable

Demonstrate how to use the nesting code formatting, and the understanding of media queries together with nesting.

In writing vanilla CSS, you can have a maximum of three levels of nested code:

- Media query
- @supports wrapper
- Selector

LESS gives you the power to nest CSS as deep as you wish. This LESS is then compiled back into the above two levels of nested code.

Here is an example:

```
// app/code/SwiftOtter/Test/view/web/css/source/_module.less

.element-selector {
  color: #000000;

  .child-selector {
    background-color: #ff0000;
  }
}
```

This is compiled into:

```
.element-selector { color: #000000; }
.element-selector .child-selector { background-color: #ff0000; }
```

Media queries also follow this idea.

```
.element-selector {
    background-color: #ff0000;

    @media screen (min-width: 750px) {
      background-color: #0000dd;
    }
}
```

This is compiled into:

```
.element-selector {
  background-color: #ff0000;
}

@media screen (min-width: 750px) {
  .element-selector {
    background-color: #0000dd;
  }
}
```

While nesting can appear very helpful, it can introduce tremendous bloat into styling. It seems to fix specificity problems but adds many extra bytes. The question that we ask is: "Are the additional selectors necessary?" In many cases, they are not. Small changes to deeply nested selectors can have significant impact on output. For instance, adding a comma and second class name to a selector five levels deep doubles the entire chain. Deep nesting also leads to hard to manage selectors that are more specific than is required and difficult to override. Instead, using a quality naming convention, such as BEM will eliminate many levels of nesting and simplify code reuse.

Describe how the `&` (Ampersand) works and its function. Describe how calculations are possible as well.

The `&` is a concatenation character and is particularly useful with writing BEM styles.

Here is an example:

```
.element-selector {
  color: #000000;

  &__text {
    font-size: 1rem;
  }

  &__call-to-action {
    background-color: #ff0000;
  }
}
```

This compiles to:

```
.element-selector { color: #000000; }
.element-selector__text { font-size: 1rem; }
.element-selector__call-to-action { background-color: #ff0000; }
```

Calculations are also possible:

```
.element-selector {
border-width: 1px + 1px;
}
```

This compiles to:

```
.element-selector { border-width: 2px; }
```

It is generally better to limit its use, though, because it can make finding the initial declaration of the style more tedious because searching is essentially useless.

Further reading:

- Operations

3.05 Demonstrate the ability to style emails

Common styles

General styles common for all emails are located in the email.less file. You have a few options to customize:

- You can customize emails by adding `_email.less`, `_email-variables.less` and so on (see the link below for the full list).
- You can always override email.less files using the standard fallback rules.

Specific styles:

- For a given module, you may specify _email.less file that contains styles specific to the emails owned by the module.
- In the template you can include CSS instructions like this:

```
<style type="text/css">
    {{var template_styles|raw}}

    {{css file="css/email.css"}}
</style>
```

Further reading:

- DevDocs: Styles for Email Templates

3.06 Explain the hierarchy of styles (lib, modules, themes)

In Magento, styles are located on different "levels". They are:

1. UI Library
2. Blank theme
3. Luma theme
4. Custom theme

In a custom theme, one may override or change any styles or variables defined above. See the general discussion about the fallback process in Section 1.2.

Note, that Magento is looking for files in the order 4-1 (so it first looks in the custom theme, then Luma, then blank), assuming that a custom theme extends Luma (otherwise Magento skips 3). For example, if Magento has to find a certain less or js file, it looks for this file in the appropriate folders of a custom theme, its parents, and UI library. See 1.2 for general discussion of the fallback process.

We will discuss the UI library below. The structure of the theme and relationship between a theme and modules will be discussed in other sections of this document. See 1.1, 1.2, and 3.1-3.3.

See 3.7 for customization of lib files.

Magento UI library usage

Demonstrate your understanding of Magento's UI library, a LESS-based library of mixins and variables for many different standard design elements on websites. How can you take advantage of the UI library? What do you have to do to enable it in your theme?

Magento has built a library of mixins to attempt to simplify the task of theming common layout components. These are found in `lib/web/css/source/lib/`.

If you have built a custom theme, these components will need to be imported with a path like:

```
@import (reference) '../../css/source/lib/lib';
```

As an example, let's add a new font onto our website design (don't take the font names seriously):

```
.lib-font-face("Trebuchet MS", "@{baseDir}fonts/trebuchet-ms", 500);
```

Another example:

```
.footer-breadcrumbs {
  .breadcrumbs();
}
```

Caution: some of the mixins are extraordinarily large. Instead of including a large mixin to accommodate your interface, we recommend exploring the option of matching Magento's selector structure in order to leverage their styles. This prevents introducing a large amount of nearly duplicate code.

Further reading:

- UI Library
- Using Custom Fonts

Which file is primarily used for basic setup of variables? Where can UI library files be found?

New variables should be placed in local theme `lib/` or local theme files. Overrides of existing variables should be declared in this file: `<theme_dir>/web/css/source/_theme.less`.

The source UI library files are located in `lib/web/css/source/lib/`.

Source: `Magento/blank/web/css/source/_theme.less`

3.07 Demonstrate the ability to implement and customize LESS library components

How can a lib file be extended?

There are multiple ways to extend a lib file. You can extend it by copying the `.less` file into your theme. For example, if you need to customize `_dropdowns.less`:

```
# Mac/Linux copy [from] [to]
cp lib/web/css/source/lib/_dropdowns.less app/design/frontend/SwiftOtter/Flex/css/sourc
```

The caveat here is that the less you copy core files, the more upgradeable the application will be.

Another approach is to implement your own custom version of the file, containing only your overrides. You can do this by adding the following file:

```
app/design/frontend/SwiftOtter/Flex/css/source/custom/lib/_dropdowns.less
```

This will not be imported by default as it is an additional file, so you will need to add the import yourself. You can copy the core import file (`web/css/_styles.less`) to your theme at the following location:

```
app/design/frontend/SwiftOtter/Flex/css/_styles.less
```

Then add the additional import. The best method to use will be dependent on the level of extension required. For example, in a situation where you want to override a single mixin within a lib file that contains multiple, it may be beneficial to just add your own custom file containing the single override. This method also helps to clearly identify where edits have been made.

Further reading:

- [A Look at CSS and Less in Magento 2]()

How can you change specific parts of the UI library?

The mixins usually have default parameters that are variables declared in the `./variables` folder. As a result, many things can be changed by setting your own values on those variable names in the `_theme.less` file. You can also override the files and make further changes.

Finally, when including a Magento UI library mixin in your own code, you can specify custom values for the many parameters that are available. The breadcrumbs UI widget (`lib/web/css/source/lib/_breadcrumbs.less`) has almost sixty variables that can be customized.

3.08 Identify the differences between client-side vs server-side compilation and how it works

Which LESS compilation options are available in Magento? How are they different?

Server-side and client-side are the two available options. This is configured in Store > Configuration > Advanced > Developer > Frontend Development Workflow.

Server-side: LESS files are compiled with a PHP LESS library. In developer mode, PHP will generate the CSS files on the fly provided there is not one already. Running `php bin/magento setup:static-content:deploy` will also compile the stylesheets.

Client-side: LESS files are compiled every page load on the client side. This results in exceptionally slow response times and a horrible flash of unstyled text.

How do they influence the developer workflow during theming?

Client-side (JavasSript based) compilation is hardly a consideration as developers should be more efficient than that. Magento provides a Grunt toolkit which will watch the source files and re-compile the output ones when changes are made. More information can be found in the DevDocs: Compile LESS Using Grunt. Further, we have had good success using Gulp from the Snowdog Frontools project and their Magento SASS port. SASS (and Gulp) compiles significantly faster than LESS (and Grunt) and Frontools provides additional features such as BabelJS support.

Explain Magento's implementation of LESS (@magento_directive)

Demonstrate the process from magento-less files via php preprocessing into real LESS files with extracted @import directives. Where can the intermediate files be found?

Magento's LESS (PHP) classes are found in the `\Magento\Framework\Css` namespace.

When loading the page, Magento looks for any `css` insertions into the `<head/>` (`Magento/Framework/View/Page/Config/Reader/Head.php`) tag in layout XML. Any `.css` file references are changed to `.less` and a search is made for a LESS file with that name. From there, Magento parses the `@imports` and `//@magento_import` instructions (`Magento/Framework/Css/PreProcessor/Instruction/MagentoImport.php`) and assembles file paths based on the fallback directories.

Magento's Luma and blank themes load in two CSS files: `styles-l.css` (desktop) and `styles-m.css` (mobile and greater). These are the entry points into the LESS file structure. These are also the output files (except that they will have a `.css` extension). All LESS files that have been imported by another LESS file are included into one of these two files.

The intermediate files are found in the `var/view_preprocessed` directory. From there, they are symlinked into the pub/static directory.

Further reading:

- Server-side Less Compilation

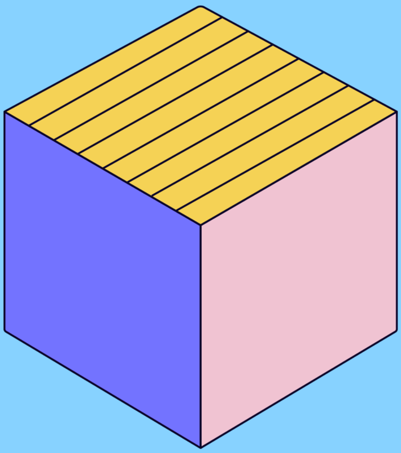
What do you have to remember, when you change a less file? Which files will be re-processed on file changes? Are the original files copied or symlinked in developer environments?

When you create a new LESS file you first need to run `bin/magento dev:source-theme:deploy`. Then (or if you just make a change to a new LESS file) you need to delete the `pub/static/[area]` directory. Refresh your browser to regenerate files (and get something to drink while you wait). Better yet, use the Grunt compiler.

TIP: Or, for even faster response, use Snowdog's SASS port and Frontools.

If you use client-side compilation, most changes are seen immediately when refreshing the browser.

The original files are symlinked (see `Magento/Framework/App/View/Asset/MaterializationStrategy/Symlink.php` and `Magento/Framework/App/View/Asset/Publisher.php`).



JavaScript (mage widgets, mage library, customer data module, Knockout templates)

Adobe Commerce Expert Frontend Developer, AD0-E720

SwiftOtter

4.01 Demonstrate the ability to initialize and call JavaScript components

Include custom JavaScript on pages

What options exist to include custom JavaScript on a page?

Most methods involve including some JavaScript or configuration directly in the page. It is then handled with Magento's special framework or RequireJS. This has the benefit of being able to apply it only to specific blocks or areas of the site. A unique option involves a RequireJS config file that is loaded from modules.

You can also include JS files into the `<head/>` tag of the rendered page. This is helpful to include external libraries.

Using the standard `<script type="text/javascript"></script>` tag. This is not recommended as Magento has no control over when this is executed. You do not have the ability to leverage any existing JS classes, and it can block rendering of the page. In some cases, this is the right solution, for example if an external bundling solution is used. For other code, use the `deferred="true"` or at least the `async="true"` attributes to the layout XML `<script>` tag so the user experience is impacted as little as possible.

Using the `data-mage-init='{ "SwiftOtter_Module/js/modal": {"configuration-value":true} }'` attribute. This special Magento attribute requests and instantiates a JavaScript module that takes two parameters: the element which hosts this attribute and the associated configuration (seen above in the ellipses). The module would look like:

```
// app/code/SwiftOtter/Module/view/frontend/web/js/modal.js

define([], function() {
  return function(element, config) {
    /* config: {configuration-value: true} */
  };
});
```

Using the Magento script tag:

```
<script type="script/x-magento-init">
  {
    ".element-selector": {
      "SwiftOtter_Module/js/modal": {
        "configuration-value": true
      }
    }
  }
</script>
```

In the above `.element-selector`, if multiple elements are found matching that selector, a new version of the module will be instantiated for each one (the module will still only be downloaded once). The selector can also be an asterisk (`*`). Contrary to the common use for the asterisk character in CSS selectors, here, it means that no elements are matched and `false` is passed to the script as the node argument.

Imperative notation:

While this is not usually the best way, it can be the easiest way to execute JavaScript on a page that depends on other libraries (i.e. jQuery) or modules:

```
<script type="text/javascript">
require([
  "SwiftOtter_Module/js/modal"
], function(loader) {
  /* … */
});
</script>
```

This is not recommended by Magento.

RequireJS Config:

You can include JavaScript on every page in an area using the `requirejs-config.js` file with a declaration as shown below. As a side benefit, it usually runs sooner than the special Magento attribute or script tag because there is less JavaScript that has to initialize before your module is loaded.

```
var config = {
deps: ['SwiftOtter_Module/js/modal']
};
```

Further reading:

- DevDocs: Include Static Resources
- DevDocs: Calling and Initializing JavaScript
- SwiftOtter: Adding Custom JavaScript
- Alan Storm: JavaScript Init Scripts
- Mage2.tv: JavaScript Fundamentals

What are the advantages and disadvantages of inline JavaScript?

The advantage is that inline JavaScript is easy to include, and it does not send an additional request. The downside, though, is that the code becomes difficult to reuse. Additionally, depending on what type of JavaScript is rendered, this code is not deferred and can block page loading.

If other places in the application need the same functionality, the script must be included there as well. This inhibits caching because the JavaScript cannot be cached separately from the main document and must be sent again every time. Also, as the complexity of the logic increases, it begins to be difficult to manage. At that point (or if you begin development knowing that it will soon get to this point), splitting your JavaScript into multiple files will ease development.

Additionally, inline JavaScript cannot be controlled with a Content-Security-Policy. This means that inline JavaScript is more likely to be exploited through Cross-Site-Scripting (XSS).

Further reading:

- Content Security Policy

How can JavaScript be loaded asynchronously on a page?

This occurs through placing functionality into `require` or `define` methods. What is the difference between these two methods? `require` executes immediately. When RequireJS is loading modules, and it comes across one that has this method, the contents are executed. `define` wraps a module that can be requested and used by other modules. As a result, it is only executed when called, such as `require(['SwiftOtter_Module/js/loader'], function(loader) { /* … */ });`

In the `<script/>` tag in layout XML, you can also apply common attributes that will be rendered on the `<script/>` tag in HTML, such as `defer` and `async`.

How can JavaScript on a page be configured using block arguments in layout XML?

The method `getJsLayout` exists in `Magento/Framework/View/Element/AbstractBlock.php`. This returns the value of the `jsLayout` array as set on the block. On the frontend, `<?= $block->getJsLayout();?>` returns a JSON string, which was built from the `JsLayout` array (see example in `Magento/Checkout/view/frontend/templates/`

`onepage.phtml`).

Using a hierarchy of `<arguments/>` in a block, we can assemble a JSON object for the frontend.

See this in action on the cart page:

- `checkout_cart_index.xml`
- `\Magento\Checkout\Block\Cart\Shipping`
- `shipping.phtml`

Further reading:

- Loading JavaScript in a Page with Layout XML

How can it be done directly in a .phtml template?

This is completed by following the examples regarding including JavaScript as described above.

4.02 Distinguish use cases for different JavaScript components

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`](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: {
                value: '${ $.provider }:${ $.dataScope }'
                // $.provider is equivalent to this.provider
            }
        },

        // 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.

4.03 Demonstrate the usage of RequireJS

Demonstrate understanding of requireJS

The idea behind requireJS is fundamental to building a JavaScript application of any size. Instead of thinking of JavaScript as a "scripting" language, it has progressed to the point of being capable of solving business requirements on the frontend (and, with NodeJS, on the backend, too).

RequireJS gives the capacity to split JS functionality into separate files or modules. This makes each file easy to read and easier to test. JS is capable of being an Object-Oriented Language, although, similar to PHP, JS can be written either way.

How do you load a file with require.js?

For the most part, requireJS follows a similar principle to Magento 2 backend development: a module never instantiates itself, and it does not directly instantiate another class. The dependencies that a module requests are provided to the requesting module.

There are already many examples above, but we will look at the specifics of directory paths and loading modules.

To instantiate a new application, do so in a rendered PHTML template. The advantage of this is that you can easily provide configuration details or translated string to the constructor.

```
<script type="text/x-magento-init">
{
  "*": {
    // this path maps to app/code/SwiftOtter/Test/view/frontend/web/js/test.js
    "SwiftOtter_Test/js/test": {
      "details": "test"
    }
  }
}
</script>
```

Tips:

- Do not add the ".js" to the module file path. The extension is assumed.
- Do add ".babel" to the module file path if you are utilizing Frontool's Babel.

Note that you can find these module names in requirejs-config.js (example: `Magento/Cms/view/adminhtml/requirejs-config.js`).

Further reading:

- Alan Storm: JavaScript Init Scripts
- Mage2.tv: Loading JavaScript in a PHTML Template

How do you define a require.js module?

```
define([], function() {
    return function(config, el) {
        console.log({el: el, config: config});
    }
});
```

Further reading:

- Defining a JavaScript RequireJS AMD Module
- Passing Arguments from PHP to Simple RequireJS JavaScript Modules

How are require.js module dependencies specified?

They are an array specified in the first parameter of the `define` or `require` methods.

How are module aliases configured in requirejs-config.js?

An alias provides flexibility in directing requireJS to the correct (or new) URL for a module. This is set up in your module's `view/[area]/requirejs-config.js`.

Aliases are configured in the `map` element in `requirejs-config.js`.

Further reading:

- Replace a Default JavaScript Component
- Aliasing RequireJS Module Files with RequireJS Config

How do you regenerate the compiled requirejs-config.js file after changes?

In developer mode, `requirejs-config.js` is regenerated on every page load. In default or production, `requirejs-config.js` is generated if it doesn't already exist. See `vendor/magento/module-require-js/Model/FileManager::ensureSourceFile()`.

Further reading:

- vendor/magento/framework/RequireJs/Config.php

How do you debug which file a requireJS alias refers to?

Start by locating where the alias originates. This is found by a simple file search. Scope can be limited to `requirejs-config.js` files if desired, or search for the alias name in the merged requirejs-config.js file in the theme directory within `pub/static/<area>`. Determine if there are any overrides for the alias. Map these files to the directory path as described above.

Demonstrate that you understand how to create and configure Magento JavaScript mixins.

Let's look at an example of how to create a mixin and override an existing module's functionality.

We need to update functionality in: `https://magento.test/static/version1/frontend/SwiftOtter/Flex/en_US/Magento_Catalog/js/gallery.js`

This file is `pub/static/frontend/SwiftOtter/Flex/en_US/Magento_Catalog/js/gallery.js`

Its source file is `vendor/magento/module-catalog/view/frontend/web/js/gallery.js`.

Add the following to your `requirejs-config.js` file:

```
// app/code/SwiftOtter/Flex/view/requirejs-config.js
var config = {
    "config": {
        "mixins": {
            "Magento_Catalog/js/price-utils": {
                "SwiftOtter_Test/js/price-utils-override": true
            }
        }
    }
};
```

Then, create this file:

```
// app/code/SwiftOtter/Flex/view/frontend/js/price-utils-override.js

define([], function() {
  var updates = {
    formatPrice: function(amount, format, isShowSign) {
      return '00000';
    }
  };

  return function(target) {
    return Object.assign(target, updates);
  }
});
```

Further reading:

- DevDocs: JavaScript Mixins
- Inchoo: How to Add Custom JavaScript in Magento 2 with RequireJS
- Mage2.tv: Customizing JavaScript jQuery UI Widgets with RequireJS Mixins
- Mage2.tv: Customizing JavaScript Objects with RequireJS Mixins
- Mage2.tv: Customizing JavaScript UI Components with RequireJS Mixins

4.04 Demonstrate the ability to implement different types of mixins

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.

This DevDocs article will get you up to speed on the specifics of building mixins: JavaScript 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 JavasScript 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: Rewrite Widget Function with Mixin.

When adding or modifying a widget, the callback inside the module should return the widget. It is passed to 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 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.

4.05 Describe how to add a translation in JS

Demonstrate the ability to use the JavaScript translation framework

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 (DevDocs: Generate a Translation Dictionary). This command collects strings that meet the translation criteria (Translate Theme Strings) 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 reading:

- Translate Theme Strings

Notes:

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

- 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?

Follow the DevDocs: Strings Added in JS Files.

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

Demonstrate an understanding of string translation in JavaScript.

The first example under each heading below shows rendering a plain string. The second example shows how to use substitutions.

UI Component templates:

```
<span data-bind="i18n: 'Shopping Cart'"></span>
<input type="text" data-bind="attr: {placeholder: $t("Email")}"/>
<translate args="'Shopping Cart'"></translate>
<span translate="'Shopping Cart'"></span>
```

UI Component configuration files: specify `translate="true"` on the element that contains text.

JS files:

require the `jquery` and `mage/translate` modules, then

```
$.mage.__('Shopping Cart')
$.mage.__("%1 items in your cart").replace("%1", numberOfItems);
```

Alternatively require only `mage/translate` and assign it to a variable named `$t`

```
$t('Shopping Cart')
```

The method names `$t` or `$.mage.__` are mandatory, otherwise the JavaScript translation string will not be included in the generated js-translation.json file.

Further reading:

- Translate Theme Strings
- Mage2.tv: Localization in JavaScript

4.06 Describe interactions between UI components

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 in the DevDocs: Linking Properties of UI Components.

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>
```

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.

4.07 Demonstrate the usage of Knockout JS

Understanding knockout framework

KnockoutJS is the framework that powers much of Magento's frontend. It is structured around the idea of reactive data: instead of the developer tracking changes, and subscribing to events, data flows through the system and updates the pertinent bindings as it arrives. It has been built on top of and abstracted quite extensively in Magento.

Further reading:

- Alan Storm: KnockoutJS Primer for Magento Developers
- Inchoo: KnockoutJS in Magento 2

How do you use knockout.js bindings?

Bindings are a way to link reactive data with HTML. They are connected with the `data-bind` attribute like:

```
<input type="text" data-bind="value: firstName "/>
```

Assuming `firstName` is an observable in the Knockout model, the value of `this.firstName` will be automatically updated whenever the input's value changes.

There are many bindings available. Here are some of the most common:

- `text` binding: sets the `innerText` of an element.
- `visible` binding: controls whether or not an element is visible.
- `if` binding: similar to `visible` but removes or adds the child nodes based on the evaluation of the property.
- `click` binding: calls the specified function when the element is clicked. It's a one-way binding (HTML to JavaScript) which differs slightly from other bindings that are bi-directional.
- `value` binding / `checked` binding / `options` binding

Magento also provides some custom bindings. All of them can be set as regular bindings in a `data-bind=""` attribute, but some may also be specified as virtual knockout elements, custom attributes, or custom elements. All of the following are more or less equivalent if used within a knockout .html template:

```
<span data-bind="text: '1. Text output'"></span>
<!-- ko text: '2. Text output' --><!-- /ko -->
<span text="'3. Text output'"></span>
<text args="'4. Text output'"></text>
```

Note that not all bindings are supported in all variations, and for some, the binding name is different (for example `data-bind="i18n: '…'"` and `translate="'…'"`).

Further reading:

- DevDocs: Custom KnockoutJS Bindings
- KnockoutJS: Visible Binding

How do you bind a ko view model to a section of the DOM with the scope binding?

The `scope` binding connects a UI component that is registered in the `uiRegistry` with an element that has already been configured. This changes that element's Knockout binding scope to use the class that represents the specified UI component.

```
<div data-bind="scope: 'estimation'">
    <p data-bind="text: description"></p>
</div>
```

In this example, the `div` tag will be bound directly to the UI component that is registered in the uiRegistry with the identifier `estimation`: `Magento/Checkout/view/frontend/layout/checkout_index_index.xml`

Due to the complexity of this, I highly recommend doing some experimenting and watching Mage2.tv's excellent series on Knockout and data binding.

Further reading:

- Mage2.tv: JavaScript
- DevDocs: Custom KnockoutJS Bindings

How do you render a ko template of a UiComponent?

The UI Components on the checkout page provide a good example regarding how to specify the template used to be rendered.

`vendor/magento/module-checkout/view/frontend/web/js/view/payment.js`

Here is a basic UIComponent and template:

```
// app/code/SwiftOtter/Flex/view/web/js/example-component.js

define([
    'uiComponent'
], function (
    Component
) {
    return Component.extend({
        defaults: {
            template: SwiftOtter_Flex/example-component',
            sampleText: "Here is some sample text."
        }
    });
});
```

```
// app/code/SwiftOtter/Flex/view/web/template/example-component

<div class="sample-component">
   <p data-bind="text: sampleText"></p>
</div>
```

Further reading:

- StackExchange: KnockoutJS Custom Template Binding

Demonstrate an understanding of the different types of Knockout observables.

Observables allow a cascading effect of functionality to happen when data changes. Typically, when we build interfaces, we receive an `input`'s `change` event and then process the data. While that works, observables remove us having to write the code to link the data change to the events that transpire. As such, they are very powerful.

The basic observable is created with a call to `this.price = ko.observable(12.42)`. To get an observables value, it needs to be called as a function without arguments (e.g.

`this.price()`). To update the value, it needs to be called as a function with an argument (e.g. `this.price(12.99)`. Any update to the value triggers any dom elements that Knockout rendered with that observable to be re-rendered. This works well for any data type except arrays. If the observable contains an array or an object, a regular Knockout observable does not trigger updates when values within the array change. For this reason, `ko.observableArray([])` should be used when working with arrays. There also are observable functions, which are called "computed observables" and are created with a call to `ko.computed(callback)` or `ko.pureComputed(callback)`.

Magento also includes the Knockout plugin ko-es5. The benefit of ko-es5 is that it allows the creation of observables that can be read and set like regular properties, while Knockout still tracks the changes under the hood.

You can use the observable's `subscribe` method to listen to changes. The real value of the observable comes out when you bind that to an element in the template like:

```
<div data-bind="text: price"></div>
```

What common ko bindings are used?

See above section for a list of common ko bindings.

Demonstrate an understanding of ko virtual elements.

Virtual elements are a way to bind a child element without having to insert a node into the DOM just to specify the binding.

```
<ul>
  <li>Please choose</li>
  <!-- ko foreach: payments -->
  <li data-bind="text: $data.name"></li>
  <!-- /ko -->
```

```
</ul>
```

Supplying the default first list item cannot be easily done when the `foreach` binding is specified on the `ul` element:

```
<ul data-bind="foreach: payments">
  <li data-bind="text: $data.name"></li>
</ul>
```

Further reading:

- KnockoutJS: Custom Bindings for Virtual Elements

4.08 Demonstrate the usage of jQuery widgets

Demonstrate understanding of using jQuery

Demonstrate understanding of jQuery and jQuery UI widgets.

jQuery is a library to assist with JavaScript development while smoothing over browser inconsistencies. It had its place several years ago while Magento 2 was in development. Here is an interesting discussion about the future and place of jQuery.

Magento uses jQuery v3.6.0:

- jQuery.com
- `lib/web/jquery/jquery.min.js`.

jQuery is famous for the `$('.element-selector').methodName(/* … */)` syntax. This snippet locates the HTML elements that match the selector and calls the fictitious `methodName` function on each. There are many other methods that can be called on the selector to modify its appearance or how it functions. jQuery is slightly less verbose than vanilla JavaScript, although this usually comes at a cost of browser performance.

jQuery UI widgets are an extension of jQuery that provides pre-built components for displaying common functionality. Here are a list of core widgets.

Further reading:

- How to Use the Widget Factory

Demonstrate understanding of how to use Magento core jQuery widgets.

Magento includes a number of jQuery widgets that are available to customize the look of the frontend. The goal is to save development time by offering pre-built solutions to solve business requirements.

These are listed below in further reading.

DevDocs has good examples for how to utilize these widgets:

- Confirmation Widget
- Accordion Widget

Further reading:

- DevDocs: jQuery Widgets
- jQuery.com: Widget Factory

How can jQuery UI Widget methods be instantiated?

- Create a `require` method or a JS module with the `define` method.
- Specify the path of the module. For the confirmation widget, it would look like:
  ```
  require(['Magento_Ui/js/modal/confirm'], function(confirmWidget) { /* … */ });
  ```
- Utilize the widget by calling the function that was imported:
  - If the widget is independent of an HTML element: `confirmWidget(/* … */)`

- If the widget affects an HTML element, you need to use the widget name as it was originally defined: `$('.element-selector').accordion({ /* … */ });`

How does variable naming/scope work with RequireJS and Magento widgets?

When a widget is defined, it will look something like this:

```
define([
  "jquery"
], function($) {
  $.widget("swiftotter.mywidget", {});
  return $.swiftotter.mywidget;
});
```

In the above code, we are exporting the newly-created widget from a RequireJS module. Here is where an important distinction is made. **When creating the widget, you assign its name to the jQuery object.** As such, that widget is callable from the jQuery selector.

However, the widget is **also** exported from the module. If you do not need to use the jQuery selector, you can import the widget and just use the name as imported.

```
define([
  "jquery",
  "SwiftOtter_Flex/js/mywidget"
], function($, importedMyWidget) {
   $(".element-selector").mywidget(/* … */);
   /**
    * The above works as it is referencing the same name as specified when it was
    * created. This is because this widget needs to be attached to HTML element(s).
    */

  importedMyWidget(/* … */);
  /**
```

```
    * Calling the variable importedMyWidget works because that is what is defined in
    * this scope.
    */
 }
```

How can you call jQuery UI Widget methods?

This answer is for widgets who are attached to an element.

```
$('.element-selector').accordion("activate");
```

The syntax is to execute the widget's name and pass, as a method argument, the name of the function you want to execute.

Further reading:

- [Accordion Widget]

How can you add new methods to a jQuery UI Widget? How can a jQuery UI Widget method be wrapped with custom logic?

There are two ways to implement this. If the business requirements dictate modifying something already in use (like the product page tabs), a different approach must be used than if you are creating something new (and have the capacity to specify a new component).

We will address modifying an existing piece of functionality first. This is done by creating a mixin-a frontend parallel to interceptors (plugins).

To override an existing Magento jQuery UI widget:

```
//app/code/SwiftOtter/Test/view/frontend/requirejs-config.js

var config = {
    "config": {
        "mixins": {
            "mage/tabs": {
                'SwiftOtter_Test/js/tabs-mixin': true
            }
        }
    }
};
```

```
// app/code/SwiftOtter/Test/view/frontend/web/js/tabs-mixin.js

define(['jquery'], function(jQuery) {
    return function(original) {
        jQuery.widget(
            'mage.tabs',
            jQuery['mage']['tabs'],
            {
                activate: function() {
                    // your custom functionality here.
                    return this._super();
                }
            }
        );

        return jQuery['mage']['tabs'];
    }
});
```

Many thanks to Alan Storm's article on this subject which makes it very clear on how to do this.

Here is an example on how to add a new method to a Magento jQuery UI widget by creating

a new widget that extends another widget:

```
// app/code/SwiftOtter/Test/view/frontend/web/js/custom-tabs.js

define([
  'jquery',
  'jquery/ui',
  'mage/tabs'
], function($) {
    $.widget('swiftotter.customTabs', $.mage.tabs, {
        doSomething: function(input) {
            alert("Nothing like a good ol' fashioned ALERT in your face. " +
                    "Input value: " + input);
        }
    });
});
```

Then, in your PHTML template:

```
// app/code/SwiftOtter/Test/view/frontend/templates/test.phtml
<div class="element-selector"></div>
<script>
require([
  'jquery',
  'SwiftOtter_Test/js/custom-tabs'], function ($) {
    $(".element-selector").customTabs();
    $(".element-selector").customTabs("doSomething", "hello");
    // this method demonstrates how to call a function on a widget
});
</script>
```

Further reading:

- Alan Storm: Modifying a jQuery Widget

- DevDocs: Use Custom JavaScript

4.09 Demonstrate the usage of JS components using Layout XML

Points to remember:

- Layout xml has nothing to do with UIComponents, or any other JS in Magento in general.
- There could be "custom" code that generates various JSON configurations to UIComponents or other Magento JS modules. The generated code usually comes from the associated block or ViewModel.
- One area where layout xml is used intensively for UiComponents configuration is checkout. Note that this is not a framework, and you can not reuse it easily in different places. This functionality is a part of checkout implementation, and we will review it in this section.

The Magento checkout is built with UIComponents 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).

We will once again recommend the terrific DevDocs which contain a list of common customizations: Customize Checkout.

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 frontend 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?

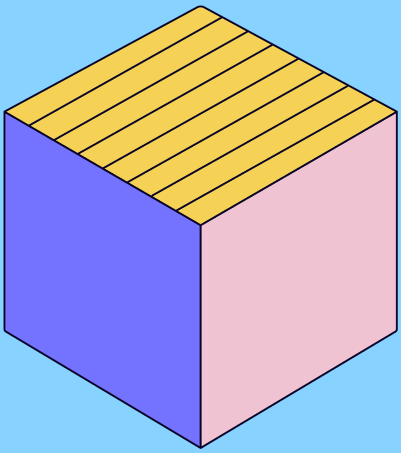
Refer to the DevDocs topic: Add a New Checkout Step

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">
```

```xml
    <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>
```



Admin Configuration and Page Builder

Adobe Commerce Expert Frontend Developer, AD0-E720

SwiftOtter

5.01 Demonstrate the ability to customize Page Builder content

The core `Magento_PageBuilder` module contains numerous content types that can be dropped into the Page Builder preview interface, and you can extend these or create your own content types with a combination of XML configuration, Knockout templates, and possibly JavaScript.

Page Builder content types

The Page Builder editing interface contains a toolbar of various types that can be dragged into the stage - content types like "Text", "Heading", "Buttons", "Banner" and "Slider". Each of these types has its own independent configuration. The starting point for a type's config is in an XML file in `view/adminhtml/pagebuilder` (e.g., `banner.xml`).

In these config files, a `<type>` is declared, with a nested hierarchy related to breakpoints, "appearances", and elements.

```
<config ...>
    <type name="banner"
          ...>
        <breakpoints>
            ...
        </breakpoints>
        <appearances>
            <appearance name="collage-left"
                        ...>
                <elements>
                    <element name="main">
                        <style ... />
                        <attribute ... />
                        ...
                    </element>
                    ...
                    <element name="content">
                        <html .../>
                    </element>
                </elements>
            </appearance>
            ...
        </appearances>
    </type>
</config>
```

The different `<appearance>` nodes for a type are essentially different potential visual layouts, each defining its collection of elements. The admin form for configuring each type will present the different appearances to choose between. An `<element>` is essentially just a container with a name, defining various attributes that can be bound into the Knockout template.

`<type>` has the following important attributes:

- `name` - The name of the type, which will also be bound to a data attribute in the output
- `label` - Label used in Page Builder
- `component` and `preview_component` - The JS components actually bound to the Knockout templates used for rendering the final output (the "master" component or the component used in the preview UI). When no unique JS is needed, these can be `Magento_PageBuilder/js/content-type` and `Magento_PageBuilder/js/content-type/preview`.
- `form` - Defines the UI component for the configuration form
- `menu_section` - Determines which menu in the Page Builder toolbar the type appears in

Each `<appearance>` and the nested `<element>` nodes have a `name` attribute to identify them. We'll explore other fields below.

Content type config XML follows the same merging behavior as other Magento XML, so this configuration can be modified in your own modules by including the same filename in `view/admihtml/pagebuilder` and using the `name` of elements you want to extend.

Wiring Forms, config and KO templates together

A content type's configuration XML file is only one piece of its definition, and in fact an intermediary piece.

The `form` attribute of `<type>` references the name of a UI form component, which is found in the core module in `view/adminhtml/ui_component`. (For example, `pagebuilder_banner_form.xml`.) These UI components define the form that allows setting configuration values for a particular instance of the type in Page Builder, and they follow the same rules and structure as any other UI components in the Magento admin.

All content type forms "extend" the UI component `pagebuilder_base_form`, which you can find in the XML file of the same name, and which defines an "Advanced" section common to all types. This section has fields for borders, margins and padding, and CSS class names. Some forms extend `pagebuilder_base_form_with_background_attributes` with common fields for a background, or `pagebuilder_base_form_with_background_video` with fields for background video.

The optional `<form>` node can be included in an `<appearance>` in the main type config XML to specify a different form per appearance, providing the opportunity for unique fields tied to an appearance.

The fields in the form UI component provide admin users with a way to set values that can then be used in the content type presentation. The main type config file in `view/adminhtml/pagebuilder` mainly functions to wire such fields to rendered style properties or attributes on each element. Two main nodes are used for this:

- `<style>` - The `name` attribute references the exact field name from the UI component, while `source` defines the camel-cased name of the CSS property the value is mapped to.
- `<attribute>` - The `name` attribute again references the field name, while `source` defines an HTML attribute name the value is mapped to. (Usually a data attribute, like `data-video-overlay-color`.)

Both nodes can also define JS components with the `converter` or `preview_converter` attribute, which will be used to modify the value before it is bound in rendering.

Another node defined somewhere in the hierarchy of basically any appearance is `<css>`, always using the `name` "css_classes" to map to the form field containing CSS classnames.

Each `<appearance>` defines the paths to two different Knockout templates, with the attributes `master_template` and `preview_template`. (For example, `Magento_PageBuilder/content-type/banner/collage-left/master`, corresponding to the template in `view/adminhtml/web/template`. Templates are in the `adminhtml` area, because they are used to render the final HTML saved in the admin.) These templates contain Knockout mark-up and can define any desired HTML structure to arrange the data from the appearance's elements. The various elements can be referenced with `data.{element-name}`. (For example, `data.main` to reference the details of `<element name="main">`.)

The unique KO bindings important to any content type are:

- `attr="data.{element-name}.attributes"` - This renders all the defined `<attribute>` data onto the HTML element.
- `ko-style="data.{element-name}.style"` - This will attach all inline styles defined with `<style>` to the HTML element.
- `css="data.{element-name}.css"` - This will output a `class` with all classnames from `<css>`.

The `master` template controls the actual saved HTML output, while the `preview` template controls how the content displays in the Page Builder preview interface. You should spend some time examining the Knockout bindings and events commonly included in preview templates.

Other important nodes in type definition XML include:

- `<static_style>` - Specifies an inline style property that doesn't need to be user-defined. This can be useful instead of stylesheet-provided rules if a property is absolutely necessary for a type to be rendered properly.
    - `<static_style source="display" value="flex"/>`
- `<html>` - Maps to a form field that provides HTML output. This can then be referenced in the KO template like `html="data.{element-name}.html"`.
    - `<html name="heading_text" converter="Magento_PageBuilder/js/converter/html/tag-escaper"/>`

Styling content types

The inline styles defined with `<style>` and `<static_style>` nodes in type configuration XML are not actually rendered inline in the HTML output. Instead, any HTML element with the `ko-style` binding in the KO template gets a data attribute with a dynamically generated value:

```
<h2 data-pb-style="SHBR1XP">This is a heading</h2>
```

For an entire content block composed of Page Builder content, a `<style>` block is then added to the beginning of the content, with the various properties implemented using an attribute selector.

```
<style>
```

```
    #html-body [data-pb-style=SHBR1XP] {
        border-style: solid;
        border-color: #0000fe;
        border-width: 1px;
        border-radius: 5px
    }
</style>
```

Further styling of content types is done with typical LESS partials in both `view/adminhtml/web/css/source` and `view/frontend/web/css/source`. These are usually organized with paths like `content-type/{type-name}/_{appearance-name}.less`. A common convention for styling Page Builder content is by using attribute selectors targeting the content type, appearance, or a particular element (corresponding to each `<element>` in the config).

```
[data-content-type='banner'] {
    [data-appearance='collage-left'] {
        [data-element='wrapper'] {
            ...
        }
    }
}
```

You can modify the appearance of a Page Builder content type in your own modules or themes using the same kinds of selectors. Note that the `#html-body [data-pb-style=...]` selector pattern used for inline styles has a specificity of 110, so you can override these simply by using a selector pattern with a higher specificity.

Breakpoints and viewports

The `etc/view.xml` file in `Magento_PageBuilder` uses the common syntax for all `view.xml` files to define an array of information related to breakpoints.

```xml
<vars module="Magento_PageBuilder">
    <var name="breakpoints">
        <var name="desktop">
            ...
            <var name="conditions">
                <var name="min-width">1024px</var>
            </var>
            <var name="options">
                ...
            </var>
        </var>
        <var name="tablet">
            <var name="conditions">
                <var name="max-width">1024px</var>
                <var name="min-width">768px</var>
            </var>
            ...
        </var>
        ...
    </var>
</vars>
```

The simplest configuration defines a named breakpoint with associated `min-width` and `max-width` values and other optional data (like `options` seen above). These named breakpoints can then be utilized in CSS and JS.

Each breakpoint config can also define additional data upgrading it to a "viewport" that is actually available in the viewport switcher in the Page Builder interface. Switching between viewports will adjust the preview stage to match the viewport constraints, giving an accurate view of the presentation at that size. (By default, "desktop", "tablet", "mobile" and "mobile-small" breakpoints exist, but only "desktop" and "mobile" are switch-able viewports.) The relevant `<var>` values and their `name` attributes include:

- "label" - The label in the viewport switcher
- "stage" - Set to `true` to make available in the stage
- "class" - Class of the "switcher" button
- "icon" - Path to an SVG icon used in the "switcher" button

To reference a named viewport in CSS, use `.{viewport-name}-viewport`. (For example, `.mobile-viewport`.)

Viewports can be interacted with in JavaScript in two ways. For the front-end, use the `matchMedia` component to trigger logic on different viewports. For the admin, listen for the event `stage:viewportChangeAfter`. In both cases, you will have access to a breakpoint object containing the details of the media conditions and any other config info attached to that breakpoint (e.g., `options`). In `Magento_PageBuilder`, see `view/base/web/js/content-type/products/appearance/carousel/widget.js` for an example. (This component is applicable both in the front-end and in the admin, which is why it uses both.)

In the content type config in `view/adminhtml/pagebuilder`, a `<form>` can be defined for a `<breakpoint>`, just like a unique form can be applied per appearance. In the corresponding form in `view/adminhtml/ui_component`, the data of any `<field>` can specify named breakpoints the value applies to:

```
<item name="config" xsi:type="array">
    <item name="breakpoints" xsi:type="array">
        <item name="mobile" xsi:type="boolean">true</item>
    </item>
</item>
```

In this way, the user-provided value for a field can be made unique to the viewport currently selected in the preview area.

As a final note, the `<var name="media">` node for a breakpoint in `view.xml` should contain a media query aligned with the `max-width` and `min-width` values. Uniquely, this "media" value is used for the `<style>` output of any viewport-specific values as described above.

```
<style>
  @media only screen and (max-width: 768px) {
    #html-body [data-pb-style=I5F99DN] {
      min-height: 100px
    }
  }
</style>
```

When defining a new viewport, in addition to the config in `view.xml` with info on icon, conditions, "media", etc., you should also add a LESS file with the appropriate styles to transform the Page Builder stage for that viewport. In `Magento_PageBuilder`, see `view/adminhtml/web/css/source/_mobile-viewport.less`.

Creating new content types

When creating a brand new content type, in addition to providing the appropriate config file, UI form component file, KO templates, and LESS file(s), also remember to create a conventional layout XML file with a name matching the pattern `pagebuilder_{form-name}.xml`. This should simply apply the appropriate `<uiComponent>`. See the core examples in `Magento_PageBuilder`, in `view/adminhtml/layout`.

When your new content type requires JavaScript functionality at runtime, you should also define the "widget" JavaScript component that corresponds with your type. (This is in contrast with the components declared in your type config file, which are used only in Page Builder to assist the output of the final HTML content that is saved.) These "widget" components are typically defined in the `base` area to be accessible from both the front-end and the preview area of the admin, and they are typically named `widget.js`. See examples in `Magento_PageBuilder`, in `view/base/web/js/content-type`.

These components should be added to the `WidgetInitializerConfig` model via `di.xml` configuration. Each separate appearance can have its own component.

```
<type name="Magento\PageBuilder\Model\WidgetInitializerConfig">
    <arguments>
        <argument name="config" xsi:type="array">
            <item name="{type-name}" xsi:type="array">
                <item name="{appearance-name}" xsi:type="array">
                    <item name="component"
                        xsi:type="string">
                        MyVendor_MyModule/js/content-type/custom-type/appearance/defau
                    </item>
                </item>
            </item>
        </argument>
    </arguments>
</type>
```

Further reading:

- Page Builder Example Modules
- Content Type Configurations
- Creating Content Types
- Extending Content Types

5.02 Describe frontend optimization

In this section, we do not cover standard frontend optimizations that are useful for every website, we only cover Magento-specific questions.

Points to remember:

- Magento compiles all less files into few css files that are included on every page.
- Magento uses RequireJS AMD concept, which implies that every js module is

encapsulated in a separate module and loaded separately which generates hundreds of connections to the server.
- CDN usually helps reduce load from the server, and sometimes improves performance, however CDNs are not supported out of the box and usually require an extension.

Show configuration and usage of CSS merging and minification

Demonstrate the primary use case for merging and minification. Determine how these options can be found in the backend. Understand the implications merging has in respect to folder traversal.

The goal of merging is to reduce the number of HTTP requests. The goal of minification is to reduce the number of bytes being transferred. Minification is valuable as it strips out whitespace which adds extra weight to the download request. Both of these provide performance boosts.

To enable / disable these settings, go into Store > Configuration > Advanced > Developer > CSS Settings.

Configure JavaScript merging and minify in the Admin UI

Note that JavaScript bundling and minifying has its pros and cons. Loading one (or multiple) large bundle is not necessarily better than loading many small scripts on demand.

What options are available to configure JavaScript minification and bundling?

The following settings are found in Store > Configuration > Advanced > Developer.

- **Merge JavaScript files:** concatenates source JavaScript files from an area together into one file to download. The idea is that this boosts performance by reducing the total number of JavaScript requests. This means a massive JS file is downloaded which can pose problems in its own way. HTTP/2 can be a better way to improve performance (or use Webpack).
- **Bundle JavaScript files:** groups the JS files into bundles. This is a similar idea to

merging but supposed to be more flexible and downloads several files. The Inchoo article below has some interesting performance statistics and turning this on could negatively impact performance.
- **Minify JavaScript files:** reduces the JS file's size by doing things like stripping whitespace and shortening variable names.

The most important is to ensure that the server is properly configured with HTTP/2 and Gzip compression enabled.

Further reading:

- JavaScript Bundling in Magento 2

How does Magento minify JavaScript?

Magento's minification of JS is a two part system. The first part is to configure RequireJS to add a `.min` suffix to files. The second part is to minify the files.

Magento tells requireJS to download minified files through the minified resolver: `vendor/magento/module-require-js/Model/FileManager::ensureMinResolverFile()`, method.

Then, Magento will minify the files as necessary. `vendor/magento/framework/App/StaticResource.php` is responsible for minifying each file.

This is where the hand-off to the JShrink library happens: `vendor/magento/framework/Code/Minifier/Adapter/Js/JShrink.php`

The minified files are saved with a `.min` suffix. As a result, all the file names are different in production. This poses a potential problem during deployment if those assets are built outside of the primary Magento database, or if the setting is changed at some point after the assets are built. If the static content is deployed with a different setting, none of the JavaScript on the site will work.

What is the purpose of JavaScript bundling and minification?

To reduce download time and make the frontend more useful and faster.

Note, that the main problem related to CDN is updating its static assets, every time they are changed (or `static-content:deploy` is executed). That's why usually we need an extension to use CDN.

Further reading:

- Using a Content Delivery Network

5.03 Customize transactional emails

How do you create and assign custom transactional email templates? How do you use template variables available in all emails? How do you access properties of variable objects (for example, `var order.getCustomer.getName`)?

Custom transactional email templates are configured in Marketing > Email Templates. These extend or inherit an existing email template. See Create and Use Custom Email Templates for a simplified lesson on creating a new transactional email template.

Once you have created a new transactional email template, you need to instruct Magento which email template to use. This is done in Store > Configuration. For order emails, go to Sales > Sales Emails. For customer emails, go to Customer > Customer Configuration.

Transactional email templates use the same directives as was discussed in 1.4. You can utilize the `{{var …}}` attribute in these email templates for any variable that was specified when initializing the email. For example, for the order notification template, the following variables are available (source `Magento/Sales/Model/Order/Email/Sender/OrderSender.php`):

- `order`: `{{var order.getCustomerEmail}}`
- `billing`: `{{var billing.getFirstname}}`
- `payment_html`: `{{var payment_html}}`, renders the HTML output from the payment block

- `store`: `{{var store.getName}}`
- `formattedBillingAddress`: `{{var formattedBillingAddress}}`
- `formattedShippingAddress`: `{{var formattedShippingAddress}}`

Additionally, in the above example, the `email_order_set_template_vars_before` is dispatched so you can modify the parameters or add more to the list.

How can you create a link to custom images from transactional email templates? How do you create links to store pages in transactional email templates?

There is not a WYSIWYG image uploader for transactional email templates. To add an image, you need to place the image in a module or a theme's `web/images` directory. You can load the image like this (see `Magento/Email/Model/Template/Filter::viewDirective()`):

```
<img src="{{view url="SwiftOtter_Test/images/your-file-name.png"}}"
    title="SwiftOtter.com"/>
```

To create links to store pages, use the `{{store url="..."}}` directive. The value in the `url` parameter is relative to the store's URL. If your store URL is `https://swiftotter.com/` and you want to link to `https://swiftotter.com/test-url`, you would add the following directive:

```
<a href="{{store url="test-url"}}" title="Test URL">Go to our test url</a>
```

Further reading:

- Email Templates
- Images in Transactional Emails

See Sections 1.4, 3.5 for more details regarding email templates in Magento.

5.04 Demonstrate the usage of admin development tools

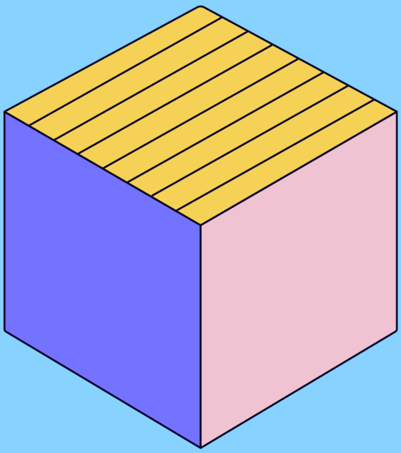
We have covered most of the admin tools already in different topics. Let's summarize them together.

Admin development tools are located in the System Configuration > Advanced > Developer.

See: DevDocs - Developer

Most important are:

- Server-side vs client-side less compilation
- Allow symlinks - which allows simpler and faster development
- Template hints
- JavaScript bundling and minifying
- CSS merging and minifying
- Sign static files



Tools (CLI and Grunt)

Adobe Commerce Expert Frontend Developer, AD0-E720

SwiftOtter

6.01 Demonstrate the usage of basic bin/Magento commands

Magento provides the command line tool out of the box. In order to run it, one issues the command:

```
$ php bin/magento <command> <arguments>
```

In order to see the list of all available commands use:

```
$ php bin/magento list
```

In order to get help on a particular command use:

```
$ php bin/magento help <command>
```

or

```
$ php bin/magento <command> -h
```

(The first option is preferable, because it executes `help` with the `<command>` as an argument which prevents the `<command>` from being executed, while the second option may accidentally execute the `<command>` if you forget the `-h`.)

For example:

```
$ php bin/magento help store:list
```

You can safely run this command to see the list of stores:

```
$ php bin/magento store:list
```

Note that some commands may cause damage if executed without understanding what they are doing.

Understand Magento console commands

How do you switch between deploy modes? What bin/magento commands are commonly run during frontend development?

To switch between deployment modes:

```
bin/magento deploy:mode:set production
```

or

```
bin/magento deploy:mode:set developer
```

To show the current deployment mode:

```
bin/magento deploy:mode:show
```

To clear all caches:

```
bin/magento cache:flush
```

To disable a cache:

```
bin/magento cache:disable full_page
```

To see the enabled or disabled state of caches:

```
bin/magento cache:status
```

To symlink the JS, LESS, and image files into the `pub/static` folder during development:

```
bin/magento dev:source-theme:deploy
```

When pushing code to production, to compile the frontend assets (note that this does not work in developer mode - Deploy Static View Files):

```
bin/magento setup:static-content:deploy
```

Other useful commands are `config:show` and `config:set`.

```
$ php bin/magento config:show
```

The above outputs all system configuration values. You can always specify variable name/ store to see a specific value. See

```
$ php bin/magento help config:show
```

for details.

```
$ php bin/magento config:set
```

This in turn can be used to set the value. It is analogous to setting the value in the admin panel.

6.02 Describe the usage of Composer commands (install, update, require, remove)

Composer is a PHP tool that is used to manage packages (dependencies). Notes to remember:

- Composer's configuration is written in the `composer.json` file.
- Information about installed versions of the packages is written in the `composer.lock`

file.
- All dependencies are installed in the vendor folder.

In what follows, we assume that composer software is installed on the server and globally available with the `composer` command.

Note that composer is using its own autoloader, see Composer Schema - Files and more generally Composer Schema - Autoload.

Further reading:

- Composer Docs: Basic Usage
- Composer Docs: Composer.json Schema

Composer install

The `install` command is used to download and install required packages (read from the `composer.json` file).

The syntax is: `composer install`

Further reading:

- Composer Docs: Install

Composer update

The `update` command pulls the latest versions of the installed packages and updates.

Syntax: `composer update`

It is also possible to update specific package(s) with `composer update vendor/package`.

Important: `update` can be dangerous, because current custom code may be incompatible with the latest versions of some packages!

Further reading:

- [Composer Docs: Update]

Composer require

The `require` will add a new record in the `composer.json` and install a new package.

Syntax: `composer require vendor/package`

Further reading:

- [Composer Docs: Require]

Composer remove

The `remove` command removes a package(s) from the `composer.json` file and then uninstalls the package(s).

Syntax: `composer remove vendor/package`

Further reading:

- [Composer Docs: Remove]

6.03 Differentiate the appropriate use case for deploy modes

Describe how to use Magento modes.

See answer above for: "How does the application behave in different deployment modes, and how do these behaviors impact the deployment approach for PHP code, frontend assets, etc.?"

What are pros and cons of using developer mode/production mode? When do

you use default mode?

See answer above for: "How does the application behave in different deployment modes, and how do these behaviors impact the deployment approach for PHP code, frontend assets, etc.?"

Default mode is enabled out of the box. It is a hybrid of both production and development modes designed to be secure but also allow for some development capabilities.

How do you enable/disable maintenance mode?

`bin/magento maintenance:enable`

`bin/magento maintenance:disable`

6.04 Define Grunt setup and usage

Points to remember:

- Task runners are nodejs modules that may automate some processes, primarily related to less (sass) compilation and simplify development process.
- Magento supports grunt out of the box, although other task runners could be used as well.
- Grunt configuration files are located in the `dev/tools/grunt/configs` directory.

Prior to configuring grunt, one must install `nodejs`, `npm` (or other node package manager) and install grunt's `node` package.

In order to configure grunt, you can specify your theme in the `dev/tools/grunt/configs/local-themes.js` file.

See Using Grunt for details.

Using grunt one may perform the following tasks:

- Compile less
- Symlink static assets
- Clean preprocessed files
- Watch changes in less files and recompile on the fly (probably the most useful feature)

See Compile LESS Using Grunt for details.

6.05 Describe additional tools that cloud provides (Fastly, downloading database, content deployment, branching using UI)

Working with a site on Adobe Commerce Cloud introduces new infrastructure, tools, and concepts into the mix.

The Fastly CDN

All production Adobe Commerce environments can and should implement full page caching with Varnish. On Cloud, the implementation via the Fastly service provides additional configuration tools and additional capabilities like image optimization.

With Fastly enabled as the full page caching application, incoming page requests are received by Fastly first, which serves a cached response if possible, without the web server's involvement and without initializing the Magento application. When cached pages expire or are invalidated, Fastly sends the request to Magento for the usual execution process.

With your Cloud plan, you will be provided with Fastly API tokens and service IDs for the Production and Staging environments. (Typically, these credentials are already configured in the Magento admin.) All settings related to Fastly are found in the admin in **Stores > Configuration > Advanced > System > Full Page Cache**. The Caching Application must be set to "Fastly CDN", and further configuration settings including the Fastly Service ID and Fastly API Token are found in the **Fastly Configuration** group in the same section.

The caching application, service ID and token can and likely should be set with environment variables instead of directly in the admin:

- `CONFIG__DEFAULT__SYSTEM__FULL_PAGE_CACHE__CACHING_APPLICATION` (value 42)
- `CONFIG__DEFAULT__SYSTEM__FULL_PAGE_CACHE__FASTLY__FASTLY_API_KEY`
- `CONFIG__DEFAULT__SYSTEM__FULL_PAGE_CACHE__FASTLY__FASTLY_SERVICE_ID`

An "Upload to VCL to Fastly" button in the Store Config section will upload the standard Magento VCL configuration that is required for Fastly to properly proxy requests.

Among the Fastly Configuraiton options is a section for Image Optimization. Enabling Fastly IO from this section will upload the proper VCL snippet to Fastly and result in the offloading of dynamic image resizing to Fastly.

Further reading:

- Adobe Commerce Fastly Documentation

Cloud environments and management

Any Cloud project comes with multiple "environments." Environments exist in a hierarchy, correspond with branches in the Cloud Git repository, and can be active or inactive. *Active* environments are also deployed into working service containers in the cloud and are accessible for browsing. The exact environment hierarchy can vary depending on the Cloud plan, but they commonly include the Production environment that is the live store, a child of that environment called Staging, a child of *that* environment called Integration, and any other environments users choose to branch from this.

Cloud users have numerous capabilities for managing environments, and many of these tasks can be accomplished either through the **Project Web Interface** via a web browser or via the standalone **Cloud CLI tool**.

Instructions for installing the Cloud CLI tool can be found in the documentation, and this tool is then used from a terminal with commands like `magento-cloud {command-name}`.

Syncing and backing up environments

Each environment has its own Git branch that can continue to diverge from its parent and children, as well as its own database instance and media store. It's often desirable to refresh an environment by syncing the code and data from its parent downstream.

Syncing can be done with the Project Web Interface or Cloud CLI tool, and code and data can be synced independently.

- **Syncing data**: The current state of the parent database and media files is copied to the target environment.
- **Syncing code**: This involves a Git operation to merge the latest upstream changes into the target environment branch.

In the Project Web Interface, a "Sync" action appears in the toolbar for any environment being viewed. The interface allows you to choose whether to sync code, data, or both.

With the Cloud CLI tool, the `environment:synchronize` command can be used for the same operation, allowing the `code` and/or `data` arguments to specify which to sync. The CLI tool also allows the `--rebase` option, which results in a rebase of the environment branch.

```
magento-cloud environment:synchronize code data --rebase
```

Snapshots are another Cloud environment feature, allowing the creation of a backup that includes the current code state, mounted volume files, and all persistent service data (including the database) for a given environment. An environment is placed in maintenance mode while a snapshot is being created, and a snapshot can be restored up to seven days after it's created.

In the Project Web Interface, creating a snapshot is done from the same toolbar where Sync is found, and snapshots can be located and restored in the environment's messages list.

With the Cloud CLI tool, snapshots are handled with the commands `snapshot:create` (to create a new snapshot), `snapshot:list` (for viewing all existing snapshots), and `snapshot:restore` (to restore a snapshot, using a snapshot ID from the `list` command as an argument).

Direct database exports can also be created with the Cloud CLI tool on your local workstation, or using the ECE-Tools package (which is a Composer dependency of all Cloud projects) directly in the Cloud environment.

- `magento-cloud db:dump`
- `php vendor/bin/ece-tools db-dump`

Further reading:

- Backup Management

Branching and merging environments

New environments can be created for staging and testing new code and configuration. It's recommended only to branch new environments from Integration.

In the Project Web Interface, while viewing the environment you wish to branch from, a "Branch" option is found in the toolbar along with the Sync and Snapshot options. Simply provide the name of the new environment. When viewing an environment that is a child of another, a "Merge" option is also found in the toolbar to merge the code branch to the parent.

With the Cloud CLI tool, the following commands are available:

- `magento-cloud environment:branch --title="Title"` to branch from the current environment
- `magento-cloud environment:checkout` to check out a particular environment branch locally
- `magento-cloud environment:merge` to merge the currently checked out

environment upstream

Further reading:

- Environment and Branch Management

Build and deployment

Several operations trigger a new build or deployment for a Cloud environment. There are three distinct phases:

- **Build**: Composer installation is done, patches are applied, etc. No services or connections are available.
- **Deploy**: Services are activated and internal network connections enabled. During this phase, the site is in maintenance mode and incoming requests are put on hold.
- **Post-Deploy**: Any tasks in this phase take place after connections to the website resume. For example, certain pages can be "warmed" in the page cache.

A full new build is triggered when new commits are pushed into the corresponding Git branch. However, other operations (like updating environment variables) can also trigger the Deploy and Post-Deploy phases. Some variables (see below) are relevant to only specific build/deploy phases.

Further reading:

- Deployment Overview

Cloud variables

There are a number of different types of variables, in various contexts, related to Cloud architecture.

One of the most notable types of variables is *environment variables* that can be set directly on environments, and in particular those that can override Store Configuration settings. The names of these variables have the format `CONFIG__{SCOPE}__{PATH}`, where `{SCOPE}` is either "DEFAULT" or a combination of a scope type and scope code (e.g., "WEBSITES_DEFAULT"). `{PATH}` is the path of the Store Config field, all uppercase with `/` replaced by `__`.

The `env:` prefix is also required for Store Config variables. Here's an example of the correct variable name to set the `general/store_information/name` Store Config setting for the "FR" store: `env:CONFIG__STORES__FR__GENERAL__STORE_INFORMATION__NAME`.

Environment variables like this will override even values locked in `env.php` or `config.php`, and child environments can inherit their parent environments' values.

In the Project Web Interface, a Variables tab is available for managing these vars, both at the project level and when navigating to "Configure environment" for a specific environment. The Cloud CLI tool features the commands `variable:create`, `variable:delete`, `variable:get`, `variable:list` and `variable:update`.

Several unique variables are set in the configuration file `.magento.env.yaml`, instead of as environment vars. A number of these are relevant to the build and deploy process.

- `CLEAN_STATIC_FILES`: If `true`, all existing static content is removed before deploying freshly deployed content.
- `SCD_COMPRESSION_LEVEL`: The gzip compression level (0 to 9) for compressing static content
- `SCD_COMPRESSION_TIMEOUT`: Maximum execution time for static content compression, in seconds
- `SCD_ON_DEMAND`: If `true`, allows static content deployment to be run dynamically at runtime, in response to static content requests
- `SCD_MATRIX`: Allows specifying a list of specific themes and locales for which to deploy static content
- `SCD_MAX_EXECUTION_TIME`: The maximum allowed execution time for static content deployment, in seconds

- `SCD_NO_PARENT`: If `true`, prevents generating static content for parent themes during build or deploy
- `SCD_STRATEGY`: Static content deployment strategy to use, when there are more than one theme/locale combinations to deploy. "standard", "quick", and "compact".
- `SCD_THREADS`: How many threads to use for static content deployment. Default value is set based on detected CPU thread count (not exceeding 4).
- `SCD_USE_BALER`: If `true`, the Baler module is run after static content deployment to scan JavaScript code and create optimized bundles.
- `SKIP_SCD`: If `true`, skip static content deployment during the build phase
- `TTFB_TESTED_PAGES`: List of pages to use for "time to first byte" tests run after deployment, the results of which are written to the cloud log
- `WARM_UP_PAGES`: List of pages for which to preload cache after deployment. See documentation for formatting examples.
- `WARM_UP_CONCURRENCY`: The number of concurrent requests to send during cache warmup operations

The following is an example of a value for `SCD_MATRIX`, which can help speed up build time by specifying only the the themes and locales that need static content generated.

```
stage:
  build:
    SCD_MATRIX:
      "Magento/backend":
        language:
          - en_US
      "Magento/blank":
        language:
          - en_US
      "Magento/luma":
        language:
          - en_US
          - fr_FR
```

Further reading:

- Override Config Settings
- Global Vars
- Build Vars
- Deploy Vars
- Post-Deploy Vars

Optimized static content deployment

The configuration of static content deployment in your Cloud project deserves special attention, because this is a key factor in the amount of downtime your site experiences during deployments.

By default, static content deployment occurs during the deploy phase when the site is in maintenance mode and unavailable for web traffic. With the right configuration details present in disk configuration, however, the process can be moved into the build phase and lead to near "zero downtime" deployments.

The site's registered themes, website and store records, the associations between stores and themes, and certain key config information can be configured on disk in `config.php`, making these key details available during the build phase when the database is not available. Here's an abbreviated example:

```
'scopes' => [
    'websites' => [
        ...
        'base' => [
            'website_id' => '1',
            'code' => 'base',
            'name' => 'Main Website',
            'sort_order' => '0',
            'default_group_id' => '1',
            'is_default' => '1'
        ]
    ],
    'groups' => [
        ...
        [
            'group_id' => '1',
            'website_id' => '1',
            'name' => 'Main Website Store',
            'root_category_id' => '2',
            'default_store_id' => '1',
            'code' => 'main_website_store'
        ]
    ],
    'stores' => [
        ...
        'default' => [
            'store_id' => '1',
            'code' => 'default',
            'website_id' => '1',
            'group_id' => '1',
            'name' => 'Default Store View',
            'sort_order' => '0',
            'is_active' => '1'
        ],
        'french' => [
            'store_id' => '2',
            'code' => 'french',
            'website_id' => '1',
            'group_id' => '1',
            'name' => 'French Store',
            'sort_order' => '0',
            'is_active' => '1'
        ]
    ]
],
'themes' => [
    'frontend/Magento/blank' => [
        'parent_id' => null,
        'theme_path' => 'Magento/blank',
        'theme_title' => 'Magento Blank',
        'is_featured' => '0',
        'area' => 'frontend',
        'type' => '0',
        'code' => 'Magento/blank'
    ],
    ...
    'frontend/Magento/luma' => [
        'parent_id' => 'Magento/blank',
        'theme_path' => 'Magento/luma',
        'theme_title' => 'Magento Luma',
        'is_featured' => '0',
        'area' => 'frontend',
        'type' => '0',
        'code' => 'Magento/luma'
    ]
],
'system' => [
    'default' => [
        'general' => [
            'locale' => [
                'code' => 'en_US'
            ]
        ],
        'design' => [
            'theme' => [
                'theme_id' => 'frontend/Magento/luma'
            ]
        ]
    ],
    'stores' => [
        'french' => [
            'general' => [
                'locale' => [
                    'code' => 'fr_FR'
                ]
            ]
        ]
    ]
]
```

The `.magento.env.yaml` file should also be configured with these values for build-phase static content:

- `SKIP_HTML_MINIFICATION`: `true`
- `SKIP_SCD`: `false`
- `SCD_STRATEGY`: `compact`

Other key variables in `.magento.env.yaml` that have an impact on optimizing static content deployment include `SCD_MATRIX`, as previously mentioned, and `SCD_THREADS`, which should be set to the maximum possible value according to the available CPU threads.

Further reading:

- [Optimized Deployment]
- [Zero Downtime Deployment]

- Static Content Deployment