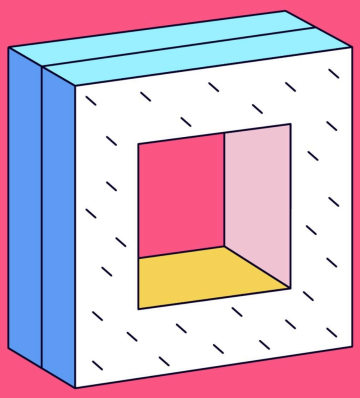


AD0-E716

Adobe Commerce Expert Developer Prep Guide

SwiftOtter

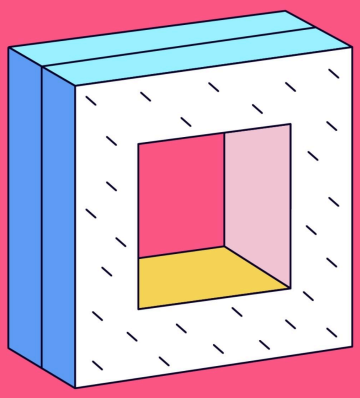
Joseph Maxwell and Vitaliy Golomoziy

Table of Contents

- Introduction
- 1.01 Demonstrate how to effectively use cache in Adobe Commerce
- 1.02 Build, use, and manipulate custom extension attributes
- 1.03 Recommend solutions for how to apply theme custom updates to product/category pages
- 1.04 Correctly apply observers, preferences, and plugins (effects of sort order)
- 1.05 Demonstrate the ability to use the configuration layer in Adobe Commerce
- Deployment configuration
- 1.06 Demonstrate knowledge of how routes work in Adobe Commerce
- 1.07 Demonstrate ability to customize Page Builder
- 1.08 Determine the effects and constraints of configuring multiple sites on a single instance
- 1.09 Describe the capabilities and constraints of dependency injection
- 1.10 Describe how to add and configure fields in store settings
- 1.11 Explain the use cases for GIT patches and the file level modifications in Composer
- 1.12 Create new commands in CLI
- 1.13 Demonstrate how to write an integration test
- 1.14 Identify Adobe Commerce security feature (CSP, escaping, form keys, sanitization, reCAPTCHA, input validation)
- 1.15 Explain how the CRON scheduling system works
- 1.16 Demonstrate the ability to load and manipulate data
- 1.17 Demonstrate the ability to use App emulation
- 2.01 Manipulate EAV attributes and attribute sets programmatically
- 2.02 Demonstrate the ability to extend the database schema
- 2.03 Demonstrate the ability to import / export data from Adobe Commerce
- 2.04 Describe how to use patches and recurring set ups to modify the database
- 3.01 Demonstrate the ability to update and create grids and forms
- 3.02 Extend Grid actions
- 3.03 Demonstrate the ability to create modifier classes

- 3.04 Demonstrate the ability to restrict access to ACL
- 4.02 Modify and extend existing Catalog entities
- 4.03 Demonstrate the ability to manage Indexes and customize price output
- 4.04 Explain how multi-source inventory impacts stock (program level)
- 5.01 Demonstrate the ability to develop new payment methods or customize existing payment methods
- Offline Payment Methods
- 5.02 Demonstrate the ability to add and customize shipping methods
- 5.03 Demonstrate the ability to customize sales operations
- 5.04 Explain how to customize totals
- 6.01 Demonstrate the ability to create new APIs or extend existing APIs
- 6.02 Demonstrate the ability to use the queuing system
- 7.01: Demonstrate knowledge of Adobe Commerce architecture/environment workflow
- 7.02: Demonstrate a working knowledge of cloud project files, permission, and structure
- 7.03: Demonstrate the ability to setup multi domain based stores on Adobe Commerce Cloud (multi domain sites mix of dev work and support)
- 7.04: Demonstrate a general knowledge of application services and how to manage them (YAML , PHP, MariaDB, Redis, RabbitMQ, etc)
- 7.05: Identify how to access different types of logs
- 7.06: Demonstrate the ability to deploy a project (Main steps of deployment)
- 7.07: Define features provided by ECE tools
- 7.08: Identify uses for ECE patches (Security breach)
- 7.09: Describe how to Maintain and upgrade ECE tools
- 7.10: Distinguish when to contact support yaml files and limitations (DIY vs Support tickets)
- 7.11: Demonstrate basic knowledge of OOTB FASTLY features configuration and installation
- 8.01: Describe how to setup/configure Adobe Commerce Cloud
- 8.02: Apply Basic Cloud troubleshooting knowledge (Hierarchy of web UI and variables, configurations precedence)

- 8.03: Demonstrate understanding of cloud user management and onboarding UI
- 8.04: Describe how to update cloud variables using UI
- 8.05: Describe environment Management using UI
- 8.06: Demonstrate understanding of branching using UI
- 8.07: Identify Adobe commerce Cloud Plan capabilities
- 9.01: Demonstrate understanding of updating cloud variables using CLI
- 9.02: Demonstrate understanding of environment Management using CLI (CLI exclusive features :activate emails, rebase environments, snapshot, db dump, local environment setup)
- 9.03: Demonstrate understanding of branching using CLI
- 9.04: Demonstrate how to troubleshoot to cloud services? (My SQL, Redis, tunnel:info)



Introduction

Adobe Commerce Expert Developer Prep Guide, AD0-E716

SwiftOtter

Introduction

You have taken the first step toward becoming a Adobe Commerce Certified Expert (Backend) 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 Adobe Commerce.

- **To get there, you might consider our prep course**. It's guaranteed to help you pass.
- As you have questions, feel free to ask in our Slack channel — the 2nd biggest Magento-focused Slack community.
- The test is 69 questions and 138 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 Shopify. 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:

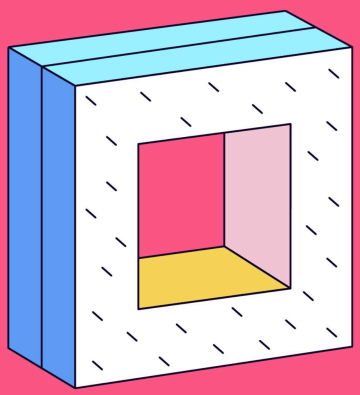
Ensure that customer data is properly cleared out. We have supplied a standard package for Magento 2 tables. You can easily create your own custom anonymizations.

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.



Objective 1

Adobe Commerce Architecture and Customization Techniques

36% of the test / 25 questions

Adobe Commerce Expert Developer Prep Guide, AD0-E716

SwiftOtter

1.01 Demonstrate how to effectively use cache in Adobe Commerce

Things to remember:

Understanding the `config`, `layout`, `block_html`, `config_webservice` and `full_page` caches and their functions is important. You can disable full page caching on a particular page by marking a block as `uncacheable="true"`.

Describe cache types and the tools used to manage caches.

Magento includes multiple types of caching to speed retrieval of CPU-consuming calculations and operations. To see a list of all cache types, run `bin/magento cache:status`.

Magento 2 includes two means of caching: server caching and browser caching.

Cache configuration is stored in `/etc/cache.xml`. Here is a list of all the `cache.xml` files in Magento 2.4:

- `module-eav/etc/cache.xml`
- `module-translation/etc/cache.xml`
- `module-customer/etc/cache.xml`
- `module-webapi/etc/cache.xml`
- `module-page-cache/etc/cache.xml`
- `module-store/etc/cache.xml`
- `module-integration/etc/cache.xml`

Further reading:

- Caching for Frontend Developers

You can clear a specific cache by using the `bin/magento cache:flush` command. For

example, the following clears the `config` and `layout` caches: `bin/magento cache:flush config layout`.

Here is a list of some of the more important caches:

`config`: Magento Configuration

The `config` cache stores configuration from the XML files along with entries in the `core_config_data` table. This cache needs to be refreshed when you add system configuration entries (`/etc/adminhtml/system.xml`) and make XML configuration modifications.

`layout`: Layout XML Updates

With Magento's extensive layout configuration, a lot of CPU cycles are used in combining and building these rules. This cache needs to be refreshed when making changes to files in the `app/design` and the `app/code/AcmeWidgets/ProductPromoter/view/[area]/layout` folders. For frontend development, we usually disable this cache.

`block_html`: Output from the `toHtml` method on a block

Obtaining the HTML from a block can also be expensive. Caching at this level allows some of this HTML output to be reused in other locations or pages in the system. For frontend development, we usually disable this cache.

`collections`: Multi-row results from database queries

This cache stores results from database queries.

`db_ddl`: Database table structure

See this file: `Magento\Framework\DB\Adapter\Pdo\Mysql`

`config_webservice`:

This stores the configuration for the REST and SOAP APIs. When adding methods to the API

service contracts, you will need to flush this one frequently.

`full_page`: **Full page cache (FPC)**

The final layer of caching in a Magento application. This HTML page output can be stored on the file system (default), database or Redis (fastest). When doing any type of frontend development, it is best to leave the FPC off. Before deploying new frontend updates, though, it is important to turn it back on and ensure that the updates do not cause problems with the cache.

How do you add dynamic content to pages served from the full page cache?

In Magento 2 FPC, pages are either cached or they are not. To make a page not cacheable, you can add the `cacheable="false"` attribute to any block on the page.

Combining speed and customization involves serving a cached page and then substituting or adding content with an AJAX request.

This DevDocs contains an article on how to do this with uiComponents.

If you prefer writing JavaScript to making these updates, the Magento API is an excellent way to load updates. REST endpoints can be configured for anonymous, customer-authenticated or admin-authenticated access.

Describe how to operate with cache clearing.

When working with the cache, assume that the requested cache entry is not present. If it is not present, run your CPU-intensive operation, and save the results into the cache. Subsequent visits to the cache should be populated and save precious computing time.

How would you clean the cache?

`bin/magento cache:clean` OR `bin/magento cache:flush`

Further reading:

- Cleaning the cache

In which case would you refresh cache/flash cache storage?

Magento recommends running cache cleaning (`cache:clean`) operations first as this does not affect other applications that might use the same cache storage. If this does not solve the problem, they recommend flushing the cache storage (`cache:flush`).

In reality, if the file system cache storage is used, you should never have multiple applications' cache storage combined. Sessions and content caching should also never share a database in Redis. Ideally, they are stored in altogether separate Redis instances. As such, flushing the cache should not have any consequences.

Describe how to clear the cache programmatically.

```
public function cleanCache(
    \Magento\Framework\App\Cache\Frontend\Pool $cacheFrontendPool
) {
    foreach ($this->_cacheFrontendPool as $cacheFrontend) {
        $cacheFrontend->getBackend()->clean();
    }
}
```

From: `\Magento\Backend\Controller\Adminhtml\Cache\FlushAll`

What mechanisms are available for clearing all or part of the cache?

`clean_cache_by_tags` event

You can dispatch a `clean_cache_by_tags` event with an `object` parameter of the object you want to clear from the cache.

From:

- `\Magento\PageCache\Observer\FlushCacheByTags::execute()`
- `\Magento\Framework\App\CacheInterface->clean()`

As seen above in “Describe how to clear the cache programmatically.”

Magento CLI console

```
bin/magento cache:flush
# or
bin/magento cache:clean
```

Manually

You can `rm -rf var/cache/*` or use the `redis-cli`, select a database index, and run `flushdb`.

How does Magento know what cached values should be cleared when a cacheable entity is saved?

In other words, let's say a product is being saved. How does Magento know which values in Redis or the file system will match?

`\Magento\Framework\DataObject\IdentityInterface`

Both the entity and the end display must implement `IdentityInterface` and return the results in `getIdentities()`. The product will typically return its ID (plus a cache tag). The end display (often a block) will return the applicable product's `getIdentities()`.

See more:

- Invalidate public content

1.02 Build, use, and manipulate custom extension attributes

Extension Attributes

Extension attributes are a new phenomenon in Magento, and one that is very welcome. Prior to Magento 2, adding data to an existing attribute, especially if that data came from another table or was calculated was difficult and error-prone.

At this point, extension attributes exert little control over how you implement, which gives you great power.

Important note: when you implement extension attributes, be aware that you must persist the data yourself. This is different than with custom attributes where the saving operations are done automatically. With extension attributes, you save the data into the database, you load it back out, and assign it to the object.

Extension attributes work with any entity that extends `Magento\Framework\Model\AbstractExtensibleModel`.

The entity's `getExtensionAttributes` method returns an auto-generated interface that contains the Camel cased getters and setters for the attribute codes specified in `extension_attributes.xml`. The setter's argument type is an interface that you create. The getter returns an instance of the interface that you created (or `null`). This interface (and thus concrete class) store the value(s) in the extension attribute.

Making an entity extensible:

- Change the model to extend `Magento\Framework\Model\AbstractExtensibleModel`
- Add two methods to the entity's service contract (interface) and model:
  - `getExtensionAttributes()`
  - `setExtensionAttributes($attributes)`

The getter return type and the setter argument type is a Magento auto-generated interface.

To determine what to place here, we will use the following interface for the entity:

`AcmeWidgets\ProductPromoter\Api\Data\PromotionInterface`

To determine the interface for the extension interface, insert `Extension` between `Promotion` (the entity type) and `Interface`: `PromotionExtensionInterface`.

Using database-joined extension attributes:

Extension attributes can be automatically loaded when loading in scalar values. Unfortunately, this behavior is mostly (only?) available out of the box for `getList` methods, like.

This is accomplished through this method:

```
\Magento\Framework\Api\ExtensionAttribute\JoinProcessorInterface
  ::process($searchResult);
```

You can replicate the following example:

- Use the `join` key.
- `reference_table` is the table to load.
- `join_on_field` is the value in the original table (`FROM`), and in this case, it's `sales_order`.
- `reference_field` is the column in the `reference_table`.
- `field`s are what you wish to join in.

```
<!-- from  -->
<config>
    <extension_attributes for="Magento\Sales\Api\Data\OrderInterface">
        <attribute code="pickup_location_code" type="string">
            <join reference_table="inventory_pickup_location_order"
```

```
                    join_on_field="entity_id"
                    reference_field="order_id">
                <field column="pickup_location_code">pickup_location_code</field>
            </join>
        </attribute>
    </extension_attributes>
</config>
```

Adding a class-based extension attributes:

The first step to adding an extension attribute is to create `etc/extension_attributes.xml`:

```
<?xml version="1.0"?>
<config>
    <extension_attributes for="Magento\Catalog\Api\Data\ProductInterface">
        <attribute code="promotions"
                type="AcmeWidgets\ProductPromoter\Api\Data\PromotionLinkInterface"
        />
    </extension_attributes>
</config>
```

- Create an interface for the type specified in the `type` parameter.
- Add getter and setter functions to interface.
- Create an object that provides a concrete implementation to the interface.
- Add the preference into `di.xml` for the interface and concrete implementation of it.
- Create plugins for the following: `afterSave` (on the entity's repository), `afterGet` (on the entity's repository), `afterGetList` (on the entity's repository). There is no magic place to set the values for the extensions as it is still a very manual process.
- The plugin should do the following:

```php
use \Magento\Catalog\Api\Data\ProductExtensionInterface as ExtensionFactory;
use \AcmeWidgets\ProductPromoter\Api\Data\PromotionLinkInterface as Promotion;

/**
 * @var \Magento\Catalog\Api\Data\ProductExtensionInterfaceFactory
 */
private ProductExtensionInterfaceFactory $productExtensionFactory;

private function injectExtensionAttributes($product): ProductInterface
{
    $extensions = $product->getExtensionAttributes();
    /** @var ExtensionFactory $extensions */
    $extensions = $extensions ? : $this->productExtensionFactory->create();

    /** @var Promotion $details */
    $details = $this->promotionsLinkFactory->create();
    $details->setValue(rand(0, 50000));
    $extensions->setPromotions($product);
    $product->setExtensionAttributes($extensions);

    return $product;
}
```

Saving the extension attribute:

The opposite of the above needs to happen. You are responsible for saving the data.

Further reading:

- Fooman: Introduction to extension attributes
- DevDocs: Creating extension attributes

1.03 Recommend solutions for how to apply theme custom

updates to product/category pages

Points to remember:

- Themes are stored in `app/design/[area]`
- `theme.xml` and `registration.php` are the only two required files.
- Overriding existing Magento or third-party module template files is as simple as replicating the directory structure in your theme.

Demonstrate the ability to customize the Magento UI using themes. When would you create a new theme?

Themes encapsulate design changes. You can extend and customize an available theme or even create one from scratch, although that is not recommended. Creating a theme usually involves copying and modifying any necessary templates, creating XML files to adjust layouts, and creating the necessary styles to make the frontend (or adminhtml area) match the design specifications.

Most merchants will have a custom theme installed. This theme could be a purchased theme or built specifically for the website.

Side note: we have used the Snowdog Blank SASS theme. It is a straight transformation of LESS to SASS and thus inherits the problems of the Magento themes. However, its compatibility with Webpack, Gulp, and Babel are worth the switch.

There are other excellent alternatives, like Hyvä (don't let the price tag scare you—it's well worth more) or Breeze by SwissUp. Luma has been deprecated and time will tell what the community settles on.

Theme directory structure

Custom themes are located in the `app/design/frontend` directory. They continue the notation of `ThemePackage_ThemeName`.

Inside this directory, the only required files are `registration.php` and `theme.xml` (`etc/view.xml` is required if there is no parent theme). The registration file is identical to a module's (in `app/code`) `registration.php`. `theme.xml` specifies the parent (fallback) theme, the theme name, and a preview image.

Further reading:

- Theme Structure

How do you define theme hierarchy for your project?

Specify the parent theme in the `parent` node of `theme.xml`.

Demonstrate the ability to customize/debug templates using the template fallback process. How do you identify which exact theme file is used in different situations?

The first step in customizing a template is to locate it. This is done by either finding the path in layout XML, using the debugger, or by enabling template hints (`bin/magento dev:template-hints:enable`). Another way is to find a unique string in the HTML (translation, tag or class). Searching the entire project directory often yields the location of the file.

Once you find the file, create a folder in your theme for the module that you copied it from. For example, if you are modifying a template from the `Magento_Catalog` module, create a `Magento_Catalog` folder inside your theme directory.

Recreate the path (including the `templates` directory) to and including the filename inside this directory that was just created.

If you are extending an existing theme, check that theme to see if the file exists there. If so, copy it from there.

How can you override native files?

Example: recently, we have been working on a project that made some extensive modifications to the customer account area. In this case, we need to modify some details shown in the Info template (account summary and newsletter subscription). Our theme is `SwiftOtter_Flow` and is found in `app/design/frontend/SwiftOtter_Flow/`.

- Locate the existing template path: `Magento_Customer::frontend/templates/account/dashboard/info.phtml`
- Because this file is coming from the `Magento_Customer` module, create the `Magento_Customer` folder: `app/design/frontend/SwiftOtter_Flow/Magento_Customer`
- Recreate the folder structure after `view/frontend/` in the existing template path: `app/design/frontend/SwiftOtter_Flow/Magento_Customer/templates/account/dashboard`
- Copy `info.phtml` from the existing template path into the new folder that was created.

1.04 Correctly apply observers, preferences, and plugins (effects of sort order)

Points to remember:

- There are three types of plugins: around, before, and after.
- Plugins only work on `public` methods.
- They do not work on `final` methods, `final` classes.
- They must be configured in `di.xml`.

Demonstrate how to design complex solutions using the plugin's life cycle.

Keeping methods to a small number of lines of code is sometimes challenging. There are those methods that seem to have to do everything.

Magento 2's idea of plugins brings a completely new idea to the table. Every public method can be intercepted, changed, or even circumvented.

There are three types of plugins: **around**, **before**, and **after**. There are some complications with the around plugin, so it is advised to use it sparingly and only when the others will not do. Before plugins modify the input arguments to a method. You can change them to any value. After plugins are used to modify the return value.

Let's say you have a complex saving operation. In this operation, you also need to validate the input data and return an error when there is a problem. You also need to respond with something more than true or false.

You can use an around plugin to validate the incoming request and cancel the save if the result is invalid. By doing this, you are applying the single responsibility principle, making your code easier to understand, debug, and test.

While plugins are often thought of as modifying core functionality, that example demonstrates that they can be useful for a broad range of applications.

How plugins work

When you create a plugin entry, Magento automatically generates a class wrapper for the plugin target. For example, if you want to modify `Magento\Catalog\Model\Product`, Magento will auto-generate the `Magento\Catalog\Model\Product\Interceptor` class. Every function inside the target class will be represented in the auto-generated interceptor class (if you add new functions to a target class, you may need to delete the auto-generated interceptor class from the `/generated` folder).

Magento then handles locating the plugins and executing them in the Interceptor class:

`\Magento\Framework\Interception\Interceptor`.

For more information, consult the plugin reference in Magento DevDocs.

Before Plugin

Example from: `\Magento\Catalog\Block\Product\ListProduct`

If you want to modify the input arguments of a method, create a before plugin. To modify the `prepareSortableFieldsByCategory($category)` method, add a method to the plugin class:

```
public function beforePrepareSortableFieldsByCategory(
    \Magento\Catalog\Block\Product\ListProduct $context,
    $category
) {
    // …
    return [$category];
}
```

The method above is run before `Magento\Catalog\Block\Product\ListProduct::prepareSortableFieldsByCategory()`. The return value for the before plugin determines the arguments going into the next plugin or the final targeted method.

After Plugin

Example from: `\Magento\Catalog\Block\Product\ListProduct`

If you need to modify the output from a public method, use an after plugin. In our example class, let's modify the `getProductPrice($product)`. As such in our plugin class, we would create:

```
public function afterGetProductPrice(
    \Magento\Catalog\Block\Product\ListProduct $context,
```

```
    $result,
    \Magento\Catalog\Model\Product $product
) {
    // …
    return $result;
}
```

The after plugin (as of 2.4) includes the input parameters in addition to the return result.

Around Plugin

The around plugin provides full control over the input and output of a function. The original function is passed in as a callback and, by standard, is named $proceed. Magento recommends against using these plugins whenever possible. This is because it is easy to accidentally alter major functions in the system by omitting a call to `$proceed()`.

Further reading:

- Magento Plugins

How do multiple plugins interact, and how can their execution order be controlled?

With every good idea comes potential downsides. Controlling how multiple plugins interact would be the problem with plugins. When a plugin is declared, the `sortOrder` attribute can be set. The lower the sort order, the sooner it will be executed in the list. The greater the sort order, the later it will be executed. This allows a degree of control over how one plugin will interact with others.

Additionally, if you need to disable an existing plugin, you can reference it by the name attribute and add the `disabled` attribute.

Plugin execution order

The basic rule for plugin execution order is this: plugins always execute based on their sort order (if no sort order is specified, sort order is determined by their module's sort order, then alphabetical order).

The Exception: When an around plugin has a lower sort order than an after plugin. Around plugins create a subroutine. See this:

- All after plugins which have a higher sort order than the around plugin are executed.
- Then the final half of the around plugin is executed.
- Finally, the after plugins that have a lower sort order are executed.

Further reading:

- Plugin Prioritization

How do you debug a plugin if it doesn't work?

First unplug it, then remove the bug.

There are a number of things that can go wrong with plugins. Here are some things to check:

- Is the `di.xml` configuration correct? Are there any syntax errors?
- Is the plugin marked as disabled?
- Do you have the correct class specified in the `<type name="...">` node? Is it the target class?
- Do you have the correct plugin class specified in the node?
- Is the class you are modifying marked as `final`? If so, plugins will not work.
- Does your plugin class have a method to modify a method on the target class?
    - `beforeMethodName` or `afterMethodName` or `aroundMethodName`
    - NOT `methodName`
- Do any of the limitations mentioned in the DevDocs apply?

One technique that has been helpful for us is to set a breakpoint in the method you want to debug. When that breakpoint has been encountered, look at the call stack to see if there are any references to an `Interceptor` class in the recent call stack.

Further reading:

- Plugins in Magento

Identify strengths and weaknesses of plugins.

Plugins are very powerful to discreetly modify functionality of existing code. They can also be used to follow the single responsibility principle (SRP) by segregating each piece of functionality to their own areas.

The greatest weakness is exploited in the hands of a developer who is either not experienced or not willing to take the time to evaluate the fallout. For example, used improperly, an `around` plugin can prevent the system from functioning.

What are the limitations of using plugins for customization?

- Plugins only work on public functions (not protected or private).
- Plugins do not work on final classes or final methods.
- Plugins do not work on static methods.
- Read more: DevDocs Plugins

In which cases should plugins be avoided?

Plugins are useful to modify the input, output, or execution of an existing method.

Plugins are also best to be avoided in situations where an event observer will work. Events work well when the flow of data does not have to be modified.

Event Observers

Ideas to remember:

- Event observers listen to events that are triggered within Magento.
- Event observers should not modify the sent data (what plugins are for).

Event observers and scheduled jobs are used to carry out tasks on data. They are an ideal way to extend Magento functionality.

Event observers and scheduled jobs carry a similar characteristic: both do not (should not) modify data as it traverses event observers. A scheduled job makes modifying the flow of data impossible while event observers still do allow it (even though it is against Magento development guidelines).

When an action occurs, an event can be triggered. Event observers listen to these events and act as a notification system. Observers implement `\Magento\Framework\Event\ObserverInterface`.

If you need to modify the data in a method, it is best to use a before or after plugin.

Demonstrate how to configure observers.

To create an event observer, create the file `events.xml` in the `etc` directory. If the event only needs to be listened to in a specific area, create an `events.xml` in that directory.

Create a class that will receive the payload from the event dispatcher. This class must implement `Magento\Framework\Event\ObserverInterface`.

How do you make your observer only be active on the frontend or backend?

Place it in the `/etc/[area]/events.xml` folder.

1.05 Demonstrate the ability to use the configuration layer in Adobe Commerce

Magento's XML configuration is split across multiple files, depending on the purpose. This

helps avoid having one very large configuration file.

Magento 2 supports configuration based on the area. This can be seen in `vendor/magento/magento-catalog/etc/` where there are the following folders: `adminhtml`, `frontend`, `webapi_rest`, `webapi_soap`. The configuration that is in these folders is only loaded if Magento is initialized in that area. For example, when browsing the admin panel, the configuration found in `frontend`, `webapi_rest` or `webapi_soap` is not loaded.

We will discuss the more important XML files found in the `/etc` folder.

`module.xml`

This is the only required configuration file. It specifies the current module's version and the module loading order.

`acl.xml`

This defines the permissions for accessing protected resources.

`config.xml`

This loads in default configuration in to Store > Configuration. This is also where configuration entries can be marked as encrypted (password).

`crontab.xml`

This identifies actions that are to occur on a schedule.

`di.xml`

This configures dependency injection for your module. This is perhaps the most frequently used file when customizing Magento. Here plugins are defined, class substitutions performed, concrete classes are specified for interfaces, virtual types setup, and constructor arguments can be specified or modified.

It is very important to familiarize yourself with the capabilities of this file.

`email_templates.xml`

Specifies email templates that are used in Magento. The template id is the concatenated XML-style path to where in system configuration template is specified.

`events.xml`

This file registers event listeners. This file can often be put into a specific area.

`indexer.xml`

Configures Magento indexers.

`adminhtml/menu.xml`

Configures the menu in the adminhtml area.

`mview.xml`

Triggers a type of event when data is modified in a database column (materialized views). These are now scheduled for reindex. The result of these operations is found in the `mview_state`. The associated values to index are stored in tables ending in `_cl`.

`[area]/routes.xml`

Tells Magento that this area accepts web requests. The route node configures the first part of the layout handle (route ID) and the front name (first segment in the URL after the domain name).

`adminhtml/system.xml`

Specifies configuration tabs, sections, groups and fields found in Store Configuration.

`view.xml`

Similar to `config.xml` but used for specifying default values for design configuration.

`webapi.xml`

Configures API access and routes.

`widget.xml`

Configures widgets to be used in products, CMS pages, and CMS blocks.

Determine how to use configuration files in Magento.

Magento's XML configuration is loaded on an as-needed basis. When di.xml is needed, Magento finds all di.xml files and merges them together.

Using a configuration file is easy: create that file in the /etc/ folder. Ideally, the file can be limited to a specific area, such as the frontend or adminhtml. Copy an existing file from another module to start with a boilerplate that works.

Which configuration files correspond to different features and functionality?

Mostly detailed above. Some less popular files:

- `address_formats.xml` : the output types for an address
- `extension_attributes.xml` : programmable EAV attributes
- `product_options.xml` : types of product options (text, file, select, etc.)
- `product_types.xml` : stores product types (simple, configurable, etc.)

Further reading:

- DevDocs Config Files

Deployment configuration

Deployment configuration is stored between `app/etc/env.php` and `app/etc/config.php`. The former is *never* committed to version control, while the latter should almost always be committed.

One of the most confusing aspects is that these files are merged and then presented as `DeploymentConfig`. The filenames are listed here. Thus, you can put any piece of information in one or the other without negative side-effects.

`env.php`

This file stores details relating to environment-specific configuration:

- Connections, like MySQL, RabbitMQ, Redis, etc.
- The `crypt_key`
- RabbitMQ `queue` details.
- Session configuration
- `MAGE_MODE` (`developer|default|production`) to determine how Magento should behave with static assets and error handling.
- `directories > document_root_is_pub`: this should be always `true`.
- `cache_types` determines which caches are enabled or disabled.
- `install` is the flag (time doesn't matter) that Magento has been installed.

`config.php`

- `modules` is a list of modules that are present with their enabled/disabled flag. Note that if a module is detected in a module directory *but it is not in* `app/etc/config.php`, this module is automatically added and enabled when running `bin/magento setup:upgrade`. As a side note, it is my experience that event

observers within a disabled module still run. **Important:** disabling a module that has a `db_schema.xml` file and a `db_schema_whitelist.json` can be catastrophic: these tables/columns are removed from the database.

- `scopes` : is a list of the websites, stores (`groups`) and store views (`stores`). Adding a website/store/store view to this list will result in its creation. This is an excellent way to keep scopes synchronized across environments.
- `themes` : this is a list of themes that are currently available. Instead of having to log into the admin (so that themes are recognized), you can add to this list. Thus, when Magento is compiling static assets, the theme will then be accessible.
- `system` : this contains all configuration values. If a value is specified in deployment config, an administrator can make no changes to this. It's locked (thus why you use the `--lock-config` or `--lock-env` flags). No matter what value is set in the respective `etc/config.xml` or `core_config_data` table, these values win.

Setting configuration

Using `etc/config.xml` , you can set a *default* value:

```
<?xml version="1.0" ?>
<config>
    <default>
        <sales>
            <order_export>
                <enabled>1</enabled>
            </order_export>
        </sales>
    </default>
</config>
```

This sets the value in `core_config_data` . The value is changeable by an administrator.

```
bin/magento config:set sales/order_export/enabled 0
```

This locks this configuration in to `app/etc/config.php`. It is no longer changeable by an administrator.

```
bin/magento config:set \
    sales/order_export/enabled \
    0 \
    --lock-config
```

Or, if this should be configured per environment, you can use the `--lock-env` flag.

```
bin/magento config:set \
    sales/order_export/enabled \
    0 \
    --lock-env
```

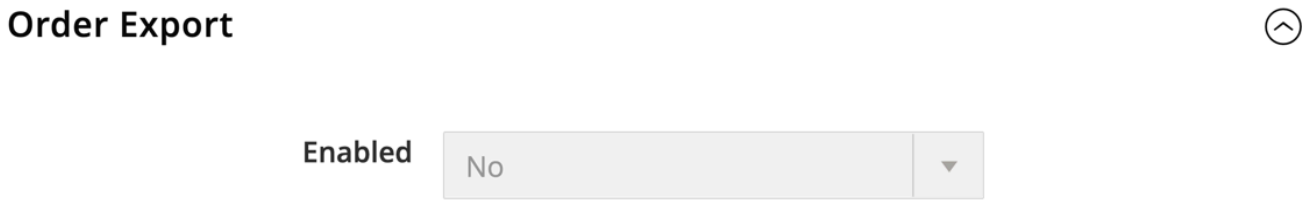
The result is:

```php
<?php
return [
    'crypt' => [
        'key' => '12345'
    ],
    'modules' => [
        // ...
    ],
    'scopes' => [
        // ...
    ],
    'themes' => [
```

```
        // ...
    ],
    'system' => [
        'default' => [
            'sales' => [
                'order_export' => [
                    'enabled' => '0'
                ]
            ]
        ],
    ]
];
```

Then, you have to run `bin/magento app:config:import`.

Here is the locked result:



Note: If you haven't populated `app/etc/config.php` yet, simply dump and commit

```
bin/magento app:config:dump themes scopes i18n
# copy down to local environment and commit
```

1.06 Demonstrate knowledge of how routes work in Adobe Commerce

Utilize modes and application initialization

Points to Remember:

- The recommended Magento entry point is `pub/index.php`
- The `default` deploy mode is the default for Magento's deploy mode feature.

Identify the steps for application initialization.

To see for yourself the path of execution, open `\Magento\Backend\Controller\Adminhtml\Auth\Login` and set a breakpoint in the `execute()` method. Clear cookies in your development website and navigate to the admin panel.

Look at the call-stack for the `execute()` method:

```
39  public function execute()
40  {
41      if ($this->_auth->isLoggedIn()) {
42          if ($this->_auth->getAuthStorage()->isFirstPageAfterLogin()) {
43              $this->_auth->getAuthStorage()->setIsFirstPageAfterLogin(true);
44          }
45          return $this->getRedirect($this->_backendUrl->getStartupPageUrl());
```

\Magento\Backend\Controller\Adminhtml\Auth › Login

Debug index.php

Debugger | Console

Frames

```
Login.php:41, Magento|Backend|Controller|Adminhtml|Auth|Login->execute()
Interceptor.php:58, Magento|Backend|Controller|Adminhtml|Auth|Login|Interceptor->__callParent()
Interceptor.php:138, Magento|Backend|Controller|Adminhtml|Auth|Login|Interceptor->Magento|Framework|Interception|{closure}()
Interceptor.php:153, Magento|Backend|Controller|Adminhtml|Auth|Login|Interceptor->__callPlugins()
Interceptor.php:26, Magento|Backend|Controller|Adminhtml|Auth|Login|Interceptor->execute()
Action.php:107, Magento|Framework|App|Action|Action->dispatch()
AbstractAction.php:229, Magento|Backend|App|AbstractAction->dispatch()
Interceptor.php:58, Magento|Backend|Controller|Adminhtml|Auth|Login|Interceptor->__callParent()
Interceptor.php:138, Magento|Backend|Controller|Adminhtml|Auth|Login|Interceptor->Magento|Framework|Interception|{closure}()
Authentication.php:143, Magento|Backend|App|Action|Plugin|Authentication->aroundDispatch()
Interceptor.php:135, Magento|Backend|Controller|Adminhtml|Auth|Login|Interceptor->Magento|Framework|Interception|{closure}()
Interceptor.php:153, Magento|Backend|Controller|Adminhtml|Auth|Login|Interceptor->__callPlugins()
Interceptor.php:39, Magento|Backend|Controller|Adminhtml|Auth|Login|Interceptor->dispatch()
FrontController.php:55, Magento|Framework|App|FrontController->dispatch()
Interceptor.php:58, Magento|Framework|App|FrontController|Interceptor->__callParent()
Interceptor.php:138, Magento|Framework|App|FrontController|Interceptor->Magento|Framework|Interception|{closure}()
Interceptor.php:153, Magento|Framework|App|FrontController|Interceptor->__callPlugins()
Interceptor.php:26, Magento|Framework|App|FrontController|Interceptor->dispatch()
Http.php:135, Magento|Framework|App|Http->launch()
Interceptor.php:24, Magento|Framework|App|Http|Interceptor->launch()
Bootstrap.php:256, Magento|Framework|App|Bootstrap->run()
index.php:102, {main}()
```

- The recommended application entry point is `pub/index.php`. Nginx or Apache should use `/pub` as the website's primary directory.
- `pub/index.php`
  - A bootstrap instance is initialized (which creates the object manager).
  - An HTTP (`\Magento\Framework\App\Http`) application is created. See concrete classes that implement `\Magento\Framework\AppInterface` to find other application types. The application is run.
- `\Magento\Framework\App\Bootstrap::run()`
  - Checks are completed (is installed, is not in maintenance mode).
  - Application is launched.

- `\Magento\Framework\App\Http::launch()`
    - Area code (frontend, adminhtml, etc.) is determined.
    - Object manager is configured for that area.
    - Front controller is created. This is based on the area.
        - Http: `\Magento\Framework\App\Http`
        - Rest: `\Magento\Webapi\Controller\Rest`
        - Soap: `\Magento\Webapi\Controller\Soap`
    - Front controller is tasked with figuring out where to direct the request.
- `\Magento\Framework\App\FrontController`
    - The list of routers (`\Magento\Framework\App\RouterListInterface`) is traversed.
    - Each router (`\Magento\Framework\App\RouterInterface`) is asked if it can match the route.
    - If it can, a `\Magento\Framework\App\ActionInterface` is returned. This action is the controller that will be executed. Controllers must implement this interface.
    - The `execute` method is run on the controller action.
    - The response from this is returned to the `FrontController`.
- `\Magento\Framework\App\Http::launch()`
    - The response is output or rendered.

How would you design a customization that should act on every request and capture output data regardless of the controller?

This is an excellent use-case for events. Create an event observer for the `controller_action_postdispatch` event.

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`

Describe front controller responsibilities.

The front controller (implementing `\Magento\Framework\App\ActionInterface`) is responsible for running business logic and returning the result of that. It could be as simple as returning an HTML layout (`\Magento\Backend\Controller\Adminhtml\System\Index`). It can also handle more complex processing, such as loading an object for editing (`\Magento\Catalog\Controller\Adminhtml\Product\Edit`).

In which situations will the front controller be involved in execution, and how can it be used in the scope of customizations?

The front controller is used for transforming the input of a url (and parameters) into an HTML

response. It is not used in the API or the console.

In most cases, if the result needed is entirely HTML or a redirect, a front controller is the best choice. If JSON is needed, while a front controller works, the API might be a better choice.

Demonstrate ability to process URLs in Magento

Many Magento urls consist of three segments:

- Front name
- Path to the action
- Name of the action Magento makes it easy to add a new routing system.

Describe how Magento processes a given URL. How do you identify which module and controller corresponds to a given URL?

Magento determines the area based on the front name (`Magento\Framework\App\AreaList::getCodeByFrontName()`). If no front name matches, the default frontend area is used.

If the request is not for the API, Magento parses the url. `Magento\Framework\App\Router\Base::parseRequest()` handles this operation. The path (the segment after the domain) is exploded with the slash as a delimiter.

Example URL: `https://storeurl.com/catalog/product/view/id/15`

- The first parameter is the module's front name. This is configured in `etc/[area]/routes.xml`. In this case, the router in `vendor/magento/module-catalog/etc/frontend/routes.xml` matches the `catalog` front name. Magento will look for a controller in the `vendor/magento/module-catalog/Controller` folder.
- The second parameter is the path to the folder that contains the action. Slashes on the file system are exchanged with underscores in the url. Magento chooses the

`Product/` inside the folder listed in the previous point.

- The third parameter is the action. Magento finds the View.php here and will run the `execute()` action.

What is necessary to create a custom URL structure?

- Create a new router (example: `\Magento\Cms\Controller\Router`) which implements the \Magento\Framework\App\RouterInterface interface. It is easiest to extend an existing router.
- Register the router (example: `vendor/magento/module-cms/etc/frontend/di.xml`)
- Return an instance of an `Magento\Framework\App\ActionInterface`

Action types Magento has introduced the capability for a controller to tag itself for what type of request it should handle:

- `POST`
- `GET`
- `PUT`
- `DELETE`
- `HEAD`
- `OPTIONS`
- `PATCH`
- `TRACE`
- `CONNECT`

Pro tip: Consider if an API request will fulfill the same requirements. Many times, an AJAX request provides a better user experience than refreshing the page. APIs are super easy to build and provide a unified mechanism for receiving and returning data.

Describe the URL rewrite process and its role in creating user-friendly URLs.

URL rewrites are a pretty face to an ugly url. The url `catalog/product/view/id/1234` is neither search engine nor user friendly. As such, Magento uses the `url_rewrite` database table to provide a means to map from a pretty url to a system url.

For example with the above url, say “super-blue-widget” has the ID 1234 in the database. In the `url_rewrite` table, you will find a row that has the following data:

- request_path: “super-blue-widget”
- target_path: “catalog/product/view/id/1234”

Magento looks for a match with the `request_path` and redirects the internal router to the target path.

The URL rewrite module contains a router which checks to see if the given route can be matched in the `url_rewrite` table. If it can, the router redirects to the route that is given in the table.

This functionality is found in the `module-catalog-url-rewrite` and the `module-url-rewrite` modules.

How are user-friendly URLs established, and how are they customized?

The user-friendly urls come from the `url_key` attributes. These attributes apply to the product and category entities.

For every category that the product is assigned to, Magento generates user-friendly urls by creating a path based on the category tree for that category. The product's url identifier is appended to this.

These values are then stored in the `url_rewrite` table.

Describe how action controllers and results function.

Action controllers implement the `Magento\Framework\App\ActionInterface` interface. Usually, they extend the `Magento\Framework\App\Action\Action` class.

Every action controller file only handles one action. This marks a difference from Magento 1 where an action controller file could handle many actions (the method name used the action url followed by “Action”).

The `execute` method must return one of the following types:

- `\Magento\Framework\Controller\ResultInterface`: this one assists in rendering HTML, JSON, or any other type of output.
- `\Magento\Framework\App\ResponseInterface`: this is for returning raw output.

How do controllers interact with another?

- A controller can forward to another route (`$this->_forward()`). The user sees no change in URL, but a new controller takes over processing the URL. This restarts the router matching loop with the new arguments.
- A controller can redirect to another URL address (`$this->_redirect()`). This returns a `Magento\Framework\App\ResponseInterface`. It returns a 30x HTTP code with the URL for the browser to redirect to.

How are different response types generated?

Types of results with default Magento 2 (must implement `Magento\Framework\Controller\ResultInterface`)):

- `Magento\Framework\Controller\Result\Json`. You should *almost never* use this. Use the web API instead. This is my opinion: returning this type is a code smell.
- `Magento\Framework\Controller\Result\Forward`
- `Magento\Framework\Controller\Result\Raw`
- `Magento\Framework\Controller\Result\Page`
- `Magento\Framework\Controller\Result\Redirect`

Inject the auto-generated Factory class into your controller. Create an instance of the class, set the values needed, and return it.

Demonstrate ability to customize request routing

Many times, business requirements accommodate Magento's preferred URL structure. This is especially true of AJAX or adminhtml requests.

If you have a small amount of customization necessary, the URL rewrite functionality can be a consideration.

Beyond that, any custom URL structure will need a custom router. Magento 2 makes it very easy to configure a custom router for your application.

How do you handle custom 404 pages?

The default Magento 404 page is a CMS page in the database. Modifications can easily be made to the CMS page.

If custom functionality is required around the generation of the 404 page, `\Magento\Cms\Helper\Page::prepareResultPage` is a public function that can be modified with plugins.

1.07 Demonstrate ability to customize Page Builder

Things to remember:

- Page Builder is included in Magento Open Source since 2.4.3. It replaces all textarea editors.
- Page Builder is written in typescript.
- Page Builder's configuration is very different from that of Magento. Pagebuilder requires configuration files to be placed in the `view/adminhtml/pagebuilder/content_type` directory of your module.

Additional resources:

- What is Page Builder?
- Page Builder examples on GitHub
- Page Builder usage

Page Builder Configuration

The crucial concept of the whole PageBuilder architecture is `content-type`. `content-type` can be seen as another "UiComponent" that defines how to manage data, UI and behavior of a certain UI element. Content-type is configured in its own file under the `view/adminhtml/pagebuilder/content_type/<type_name>.xml` (Example).

Magento merges these XML files in the typical Magento way. You must ensure that you specify the key attribute (usually `name`) that is used for merging.

The key configuration settings for the content type are:

- `name` - used for merging
- `component` - the Javascript module used for rendering, usually the standard one
- `preview_component` - the Javascript module that is used in admin panel for preview
- `master_component` - usually the standard Javascript module that is used for rendering
- `form` - UiComponent form that is used for content-type's editing
- `menu_selection` - a string that defines where in the admin panel (Page Builder menu) the content-type appears

The next important concept is appearance. Appearance defines how the data is placed on the element.

PageBuilder Customizations

There are few common customizations related to the PageBuilder.

- Enable/disable Page Builder in a given textarea input. This is done in UiComponent's

xml file.

- Hide the PageBuilder's editor and replace it with the “Edit with Page Builder” button.
- Upgrade PageBuilder's content type. This requires a special data patch.

Further reading:

- Use Page Builder for product attributes
- Upgrade content types

1.08 Determine the effects and constraints of configuring multiple sites on a single instance

Things to remember:

- Magento can be deployed in a multi-websites mode to a single host. Magento will differentiate websites based on the env variables: `MAGE_RUN_TYPE` and `MAGE_RUN_CODE`
- The architecture with many websites (no matter if there is a single host or not) will cause problems with price indexing, since prices have website scope
- Configuration parameters have different scopes, and you may set the value for each store/website using CLI, Data Patch or `<MODULE>/etc/config.xml` file

Additional resources:

- Multiple websites or stores
- Set configuration values

1.09 Describe the capabilities and constraints of dependency injection

Points to remember:

- Dependency injection is a means of giving a class what it needs to function.
- `ObjectManager` is Magento's internal object repository and should rarely be directly accessed.
- Dependency injection makes testing easier, an application more configurable and provides options for powerful features such as plugins and virtual types.

Magento is very customizable. Magento's Dependency Injection concept embraces that and allows a great deal of control.

Dependency Injection is literally injecting what a class' dependencies into the constructor or setter methods (Magento uses the constructor).

Alan Kent has a great article about dependency injection and its benefits:

The beauty of dependency injection is that it is very easy to see what your class needs are at a glance. You build your class around the class or interface that you inject. You do not care what class or interface is injected.

Describe Magento's dependency injection approach and architecture.

Magento's dependency injection framework is unique in that it is very automatic. Many other frameworks require at least some level of configuration to get going. Magento provides ways to customize and adjust dependency injection on the fly as well.

Magento uses constructor injection: that is, all of the dependencies are specified as arguments in the `__construct()` function.

Before we continue, I should note that it is very poor practice to directly use the `ObjectManager` —the primary class that handles dependency inject. It is against Magento standards to do this (exception for only a few cases). See more here.

Here is a sample constructor:

```php
<?php
namespace AcmeWidgets\ProductPromoter\ViewModel;

use \AcmeWidgets\ProductPromoter\Service\Product as ProductService;

class Details extends \Magento\Framework\View\Element\Block\ArgumentInterface
{
    private ProductService $productService;

    public function __construct(
        ProductService $productService,
        array $data = []
    ) {
        $this->productService = $productService;
    }
}
```

We enter the class type and an argument name into the constructor.

Magento's DI container holds a list of objects. Each time one is created, it is added into this container. Whenever an object is requested, it is loaded from this container. This follows the idea of a singleton (similar to `Mage::getSingleton`). In PHP, objects are references. Unless you clone the object, anywhere you pass the object (as an argument) that same object is referenced.

For objects that need to be new every time they are used, Magento uses factories. To use a factory, specify the class or interface name and append Factory to the end. Like: `\Magento\Catalog\Api\Data\ProductInterfaceFactory`. Magento will find the preference for the `ProductInterface` (default is `\Magento\Catalog\Model\Product`) and create a factory for that class. The factory has one public method, `create()`. Calling this method creates a new instance of the desired class. Magento creates these classes automatically (check this out). If you want, you can create factory classes. Here is an example: `Magento\Paypal\Model\IpnFactory`.

For objects that might be time intensive to load, Magento provides proxies (generated here). A proxy lazy loads the class. To utilize this, specify a class like `\Magento\Catalog\Model\Product\Proxy` in the constructor.

How are objects realized in Magento?

Since dependency injection happens automatically in the constructor, Magento must handle creating classes. As such, class creation either happens at the time of injection or with a factory.

Class creation at the time of injection

A great way to watch this step-by-step is to set a breakpoint in `\Magento\Framework\App\Router\Base::matchAction`, on the line that contains `$this->actionFactory->create()`.

The first step in the process is the object manager locating the proper class type. If an interface is requested, hopefully an entry in di.xml will provide a concrete class for the interface (if not, an exception will be thrown).

The deploy mode (`bin/magento deploy:mode:show`) determines which class loader is used.

Developer: \Magento\Framework\ObjectManager\Factory\Dynamic\Developer Production: \Magento\Framework\ObjectManager\Factory\Dynamic\Production

The parameters for the constructor are loaded. Then those parameters are recursively parsed. Not only are the dependencies for the initially requested class loaded, but dependencies of dependencies as well.

A metaphor of this would be a tree. In a tree, you have the trunk and then the branches. The trunk would represent an object type. But that object has dependencies, which continue splitting and going up the tree. Eventually, you have all the branches and all the leaves representing all of the classes (dependencies) that your class needs to perform its functions.

Class creation with a factory

To see Magento auto-created factories, look in the /generated folder. If you need to create a custom factory, feel free to copy one of these as boilerplate, changing the namespace, class name and likely the parameters of the `create` function.

Further reading:

- Dependency injectiion

Why is it important to have a centralized process creating object instances?

Having a centralized process to create objects makes testing much easier. It also provides a simple interface to substitute objects as well as modify existing ones.

Identify how to use DI configuration files for customizing Magento.

You should be very familiar with `di.xml` and how to use it.

Plugins

Plugins allow you to wrap another class' public functions, add a before method to modify the input arguments, or add an after method to modify the output.

When a plugin targets a given class, that class is automatically extended (here's how). This intercepted class is then used any time the original class is requested.

Example: `vendor/magento/module-catalog/etc/di.xml` (search for “plugins”)

Preferences

Preferences are used to substitute entire classes. They can also be used to specify a concrete class for an interface. If you create a service contract for a repository in your `/Api` folder and a concrete class in `/Model`, you can create a preference like:

```
<preference
```

```
    for="AcmeWidgets\ProductPromotor\Api\PromotionRepositoryInterface"
    type="AcmeWidgets\ProductPromotor\Model\PromotionRepository" />
```

Further reading:

- `di.xml` file reference

Virtual Types

A virtual type allows the developer to create a new class that extends an existing concrete class. While doing so, you can specify or change the input arguments.

We have found this to be rarely useful and can create confusion as to where this class is created.

Further reading:

- DI configuration
- Alan Storm's Object Manager Deep Dive

Argument Preferences / Constructor Arguments

It is possible to modify what objects are injected into specific classes by targeting the name of the argument to associate it with the new class.

In Magento 2, you can inject your custom class into any other classes constructor in `di.xml`.

Example:

```
<config>
    <type name="AcmeWidgets\ProductPromoter\Algorithms\Primary">
        <arguments>
            <argument name="basedOn" xsi:type="string">sort_order</argument>
```

```
        </arguments>
    </type>
</config>
```

Further reading:

- Constructor Arguments

How can you override a native class, inject your class into another object, and use other techniques available in `di.xml` (such as virtualTypes)?

Overriding a native class: use a `<preference/>` entry to specify the existing class name (preceding backslash \ is optional) and the class to be overridden.

Inject your class into another object: use a `<type/>` entry with a `<argument xsi:type="object">\Path\To\Your\Class</argument>` entry in the `<arguments/>` node.

1.10 Describe how to add and configure fields in store settings

Define basic terms and elements of system configuration XML, including scopes. How would you add a new system configuration option? What is the difference in this process for different option types (secret, file)?

Store configuration XML is stored in `etc/adminhtml/system.xml`. The resulting values that the admin specifies are stored in `core_config_data`. The following will cover the primary elements found in the `system.xml` file.

All configuration lives within the root node (`<config/>`) > `system`.

Tabs `<tab/>`

Example: `vendor/magento/module-backend/etc/adminhtml/system.xml`

```
<tab id="general" translate="label" sortOrder="100">
<label>General</label>
</tab>
```

Sections `<section/>`

These are the items within the tab accordion on the left sidebar.

Example: `vendor/magento/module-backend/etc/adminhtml/system.xml`

Important attributes

- `id` : the name of this group. This will be used to formulate the store configuration path in `core_config_data` and in retrieving this value.
- `showInDefault` : whether this section is visible or not in the default scope (no store is selected)
- `showInWebsite` : whether this section is visible or not in the website scope
- `showInStore` : whether this section is visible or not in the store scope

Available children

- `class` : CSS classes to apply.
- `label` : Title of the section (notice the attribute `translate="label"` in the parent `section` tag).
- `tab` : Which tab ID this section belongs to.
- `resource` : ACL path for this particular section (not specifying a resource means that

all admins can access this).

- `group` (1+ entries): the groups / accordions presented on the right side after clicking into this section.

Groups `<group/>`

Collapses a list of fields into one row (showing the group's label) to make browsing easier.

Important attributes

- `id` : The name of this group. This will be used to formulate the store configuration path in `core_config_data` and in retrieving this value.
- `showInDefault` : Whether this section is visible or not in the default scope (no store is selected).
- `showInWebsite` : Whether this section is visible or not in the website scope.
- `showInStore` : Whether this section is visible or not in the store scope.

Available children

- `label` : Title of the section/
- `field` (1+ entries): Describes a field's configuration.

Fields `<field/>`

This is the destination for system configuration. A field allows input by an administrator that will be saved into the database.

Important attributes:

- `i* d` : The name of this group. This will be used to formulate the store configuration path in `core_config_data` and in retrieving this value.
- `type` : One of `text` , `wysiwyg` , `textarea` , `select` , `multiselect` , `obscure` . This

can be blank if you specify the `frontend_model` node.

- `showInDefault` : Whether this section is visible or not in the default scope (no store is selected).
- `showInWebsite` : Whether this section is visible or not in the website scope.
- `showInStore` : Whether this section is visible or not in the store scope.

Available children

- `label` : The field's title.
- `comment` : An explanatory note to describe the field.
- `source_model` : Specifies a class that implements `Magento\Framework\Option\ArrayInterface` . This provides a list of options to select or multiselect.
- `frontend_model` : Specifies a class that extends `Magento\Config\Block\System\Config\Form\Field` . This is how you display a custom input field in store configuration.
- `backend_model` : Changes or adjusts data coming to / from the system. The `obscure` field type commonly uses this.
- `depends` : Makes the visibility of this field dependent on the setting of another field.
- `validate` : A CSS validation class (example: `validate-email`).

Describe system configuration data retrieval. How do you access system configuration options programmatically?

Magento provides two layers for store configuration: the defaults (specified in `etc/config.xml`) and the options stored in the database.

The data is retrieved in `Magento\Framework\App\Config` by the `Magento\Config\App\Config\Type\System` configuration type.

Accessing store configuration.

Inject `Magento\Framework\App\Config\ScopeConfigInterface` into a class needing this configuration information. Call the `getValue` (for raw value) or `isSetFlag` (for boolean value) method to retrieve the value needed.

1.11 Explain the use cases for GIT patches and the file level modifications in Composer

Points to remember:

- GIT patches distributed as .diff files - one per module.
- You can apply a patch using the console patch command or via Composer.
- There is an in-built tool to apply Quality Patches.

Quality Patches Tool

This is an additional tool created by Magento to simplify managing Magento-provided patches. The tool is an executable file located in the `./vendor/bin/magento-patches` folder.

- Install the tool using: `composer require magento/quality-patches`. The tool is located in `vendor/bin/magento-patches`.
- Available commands are: `status`, `apply`, `revert`.
- To fetch new patches you have to update the tool with `composer update magento/quality-patches`.
- The tool does not clean the cache or update the database.

Important: after upgrading to a new version, one has to manually verify which patches needs to be re-applied!

Further reading:

- MQP patches

Applying a patch via composer or command-line.

To apply the patch via command-line just download the .diff file and run the "patch" command. Note, there is nothing in this case that manages which patches have been applied.

To apply patches using composer, install cweagans/composer-patches package. After that you can edit your `composer.json` file and apply a patch using `composer update`. See documentation for more details.

1.12 Create new commands in CLI

Things to remember:

- The Magento CLI is based on the Symfony Console component.
- It is important to familiarize yourself with these commands.

Declaring a new CLI command

In order to create a new CLI command, add your command to the `Magento\Framework\Console\CommandListInterface` in the `di.xml` file.

```
<config>
        <!-- ... -->
        <type name="Magento\Framework\Console\CommandListInterface">
            <arguments>
                <argument name="commands" xsi:type="array">
                    <item name="commandexample_somecommand" xsi:type="object">
                    SwiftOtter\CommandExample\Console\Command\SomeCommand
                </item>
                </argument>
            </arguments>
```

```
        </type>
        <!-- ... -->
</config>
```

Everything else is done inside the PHP class that represents the command. The class must extend `Symfony\Component\Console\Command\Command` class. Here's a simple example.

Developing a new CLI command.

There are a few things that are important to know.

1. How to operate with arguments.
2. How to manage help.
3. Managing an area.
4. Accessing input and output.
5. CLI bootstrap.

Arguments. Usually our commands require arguments. Arguments can be mandatory or optional. Some arguments may have default values. In addition to that, Symfony provides a functionality to shortcut the arguments.

For adding an option, use the `$this->addOption()` method, which is better to place inside of the protected `configure()` method.

Here is the declaration of the addOption method in the `Symfony\Console\Command\Command` class:

```
public function addOption(
  $name,
  $shortcut = null,
  $mode = null,
  $description = '',
```

```
    $default = null
  );
```

The definition is self-explanatory, except the $mode parameter. Usually you import `Symfony\Component\Console\Input\InputOption` class and use one of `InputOption::VALUE_OPTIONAL` or `InputOption::VALUE_REQUIRED`.

Help

For managing help, use the `configure()` method described above and call `$this->setName()` and `$this->setDescription()`.

`$this->setName()` receives a parameter which indicates what is the command itself. For example, `some:command`. Note you can also set the name via di.xml.

Managing an area

By default, Magento does not initiate any area when launching CLI commands. In case you need some area in your code, you should import `Magento\Framework\App\State` and execute `setAreaCode` on it.

Accessing input and output

Since console commands extend `Symfony\Console\Command\Command`, the way to work with input and output is dictated by Symfony. In this case, we must implement the `execute()` method with two parameters: `InputInterface` and `OutputInterface`. `InputInterface` has the method `getOption($name)` which returns a value of a CLI argument and `OutputInterface` has the method `writeln()` which outputs a text to the console.

CLI bootstrap

It is important to keep in mind that the initial phase of application execution (bootstrap) is different for CLI commands compared to that of cron commands or HTTP requests. The application class is `Magento\Framework\Console\Cli` which extends `Symfony\Component\`

`Console\Application`. As a bootstrap, it uses `Magento\Setup\Application`.

Further reading:

- CLI naming guidelines
- How to build CLI commands

1.13 Demonstrate how to write an integration test

Things to remember:

- The integration test may use some pieces of the system (like database or filesystem).
- Magento widely uses annotations for integration tests.
- You should create your objects via `Magento\TestFramework\Helper\Bootstrap::getObjectManager()`.
- Use the standard `\PHPUnit\Framework\TestCase` so you can use mocking and other features of PHPUnit.
- Don't test *what happens*. Instead test that given provided inputs, you will receive expected outputs. This is known as blackbox testing.
- It is very difficult to test classes that do too much.

Configuring integration tests

Configuration files for integration tests are located at `dev/tests/integration`. Let's start with configuring the connection to the test database. First, one has to create a database, import data into it (using `mysqldump` or other technique) and configure the connection in the `dev/tests/integration/etc/install-config-mysql.php`.

Magento also allows you to pre-specify system config values (like the ones from `core_config_data`) in the file `dev/tests/integration/etc/config-global.php` See this for more details.

Another file that you may find useful is `dev/tests/integration/etc/di/preferences/`

`ce.php` (for Magento Open Source). It has pre-defined preferences for some classes. It is also a good source of information to understand which Magento classes are overridden with the Magento TestFramework.

Next important configuration file is `dev/tests/integration/phpunit.xml`. You may want to copy it for your project. There are three things to configure there.

- Testsuites in the node.
- PHP configuration. Here you can configure `includePath` and various constants that affect test execution. For example, `TESTS_MAGENTO_MODE` defines the mode in which tests are executed.
- Listeners. This is not something very important, but you may want to remove the `Magento\TestFramework\Event\PhpUnit` listener to speed up the execution.

Creating objects

The constructor injection mechanism does not work for test classes, and you have to instantiate your objects via `Magento\TestFramework\Helper\Bootstrap::getObjectManager()`. It gives you a "real" class. In case you need a stub or a mock, just use PHPUnit functionality.

Annotations

Annotations are critical to write integration tests with Magento; they are well described in the documentation.

Here we emphasize some of them.

- `@magentoDataFixture` - defines a file or method that provides the data needed for a test. Loaded files are relative to `dev/tests/integration/testsuite`.
- `@magentoAppArea` - defines the area to be used in the test.
- `@magentoAppIsolation` - isolates the app (like sessions between tests).

- `@magentoDbIsolation`: keeps each test within a transaction and rolls back at the completion. If there is an error in the processing, you will see a MySQL error about transactions not being committed. This mode is automatically enabled if `#magentoDataFixture` is used.

All annotations are important, so please check the manual.

Running integration tests:

You can run the test using the command: `phpunit -d memory_limit=1G -c phpunit-custom.xml -testsuite "My Test Suite"`. It is also possible to specify the path to the folder (for example, in order to only execute one test file or files from a single module).

1.14 Identify Adobe Commerce security feature (CSP, escaping, form keys, sanitization, reCAPTCHA, input validation)

Things to remember:

- Form keys should be used for all forms.
- Every string literal has to be escaped. There are different escaper methods for different types of string in Magento.
- Captcha can be configured in the admin panel and in the Google panel (outside of Magento).
- Magento has the whole framework for client-side input validation in form fields.

Escaping

You should escape any data entered by a customer or rendered on the page in order to prevent an XSS attack. Magento has various escapers for different purposes. There is an `$escaper` variable available in the .phtml template that has various escaping methods like:

`escapeHtmlAttr`, `escapeHtml`, `escapeUrl`, `escapeJs`, `escapeQuote` and others (see `Magento/Framework/Escaper.php`).

Input validation

In order to validate a user's input into a form, Magento provides `mage/validation` and `mage/validation/validation` components which include various validation rules (see Form Validation). You can also add a custom rule using `$.validator.addMethod` (see Custom Validation).

In order to enable validation framework for a given form use: `<form id="my-form" data-mage-init='{"validation": {}}'>`.

Then you can use various techniques to validate individual fields: `<input id="field-1" ... data-validate='{"required":true, "minlength":10}'/>`

See Custom Form Validation for more details.

Note that UiComponents forms in admin have a node in which you may configure validation for a field (either custom or native).

Content security policies

Magento supports CSP-specific headers (defined in the Magento_Csp module). There are two modes in which Magento CSP works—report-only (default) and restrict mode (in which Magento acts on policy violation). See `Magento/Csp/etc/config.xml` for details on policy configuration. The typical operation that you may need in your custom module is adding a domain to a whitelist. This can be done with a special file `etc/csp_whitelist.xml` (of your module).

Here is an example from `Magento_Paypal/etc/csp_whitelist.xml`:

```
<csp_whitelist>
```

```
<policies>
    <policy id="img-src">
        <values>
            <value id="paypal_analytics" type="host">t.paypal.com</value>
            <value id="www_paypal" type="host">www.paypal.com</value>
            <value id="paypal_objects" type="host">www.paypalobjects.com</value>
            <value id="paypal_fpdbs" type="host">fpdbs.paypal.com</value>
            <value id="paypal_fpdbs_sandbox" type="host">fpdbs.sandbox.paypal.com<
        </values>
        ...
```

Another important aspect is the `$secureRenderer` variable that is available in a .phtml template (or `$this->secureRenderer` property in a block) which allows whitelisting for inline scripts and styles.

Further reading:

- DevDocs: Content Security Policies

Other security aspects

Here are PHP functions that are not recommended for use by security best practices.

Avoiding different types of attacks:

- Mass Assignment
- Server-side Request Forgery
- Authorization
- Cross-site request forgery
- Working with files

1.15 Explain how the CRON scheduling system works

Demonstrate how to configure a scheduled job

To execute a specific action on a schedule, you need to setup the `crontab.xml` file. This file always resides in the `/etc` folder (not in `/etc/[area]`).

The basic syntax looks like:

```
<config>
    <group id="default">
        <job name="job_name_goes_here"
                instance="AcmeWidgets\ProductPromoter"
                method="execute">
            <schedule>*/10 * * * *</schedule>
        </job>
    </group>
</config>
```

The above represents the most basic possible cron configuration.

Note that to disable a cron job, simply specify an invalid `schedule`.

Group

Magento allows you to group cron activity together, making logical groups of functionality. For most scheduled activity, use the `default` group. Magento also does provide `index` group. You can set up configuration options for groups in `cron_groups.xml`.

Job

Configuring the job is simple:

- assign a unique name

- specify the class
- define a method
- set a schedule (using regular crontab schedule notation)

1.16 Demonstrate the ability to load and manipulate data

Points to remember:

- Familiarize yourself with the methods in repositories (`getById()`, `getList()`, `save()` and `delete()`).

Describe repositories and data API classes. How do you obtain an object or set of objects from the database using a repository?

Repositories represent a mechanism that handles data operations. They are designed to be used in the API or backend code.

Since repositories are written against an interface, a developer is able to change out complete implementations of accessing and modifying data with little effort.

For example, the `ProductRepository` class is written to implement the `\Magento\Catalog\Api\ProductRepositoryInterface` interface. By satisfying this interface, the entire `ProductRepository` could, in theory, be changed. Reality is a different story.

To get an object from the database, use the `getById()` method.

To get multiple objects from the database, use the `getList()` method.

Note, that repositories usually return a data object (or an array of data objects) not a model. In some cases, like the `catalog_product` entity, they are the same, but in others (like the customer entity) they could be different.

How do you configure and create a SearchCriteria instance using the builder?

How do you use Data/API classes?

- Inject an instance of the `\Magento\Framework\Api\SearchCriteriaBuilder` class.
- Call the necessary methods on the `SearchCriteriaBuilder` instance to filter (`addFilter()`) or sort (`addSortOrder()`) the final result.
- Call `getList` on the repository and pass in the created SearchCriteria.

Further reading:

- Searching With Repositories

Describe how to create and register new entities. How do you add a new table to the database? Describe the entity load and save process.

This answer covers simple entities and not EAV, which we discuss in the next segment.

Adding a new table happens in `app/code/AcmeWidgets/ProductPromoter/Setup/InstallSchema` or `app/code/AcmeWidgets/ProductPromoter/Setup/UpgradeSchema`. The `install()` or `upgrade()` method is called when `bin/magento setup:upgrade` is run in the CLI. The first parameter is an object that contains a connection to the database. As such, it is easy to create tables.

Entity load process

Entity resource models extend `\Magento\Framework\Model\ResourceModel\Db\AbstractDb`. This class contains the functionality that is used to load and save objects to and from the database. Look at the `load()` method in this class:

- The resource model's `beforeLoad()` method is called. This is a great plugin point.
- The load select is generated.
- The row is fetched.
- The object is hydrated.
- The object is returned.

Entity save process

The action happens in the `save()` method:

- A transaction is started.
- If nothing has been modified in the entity, the transaction is commit, and the method returned.
- The resource model's `beforeSave()` method is called. This is a great plugin point.
- If saving is allowed, the database representation is updated or created (based on whether the ID returns an integer or not).
- While there appears to be a bug whereby the resource model's `afterSave()` is never called, this would be the point where `afterSave()` is executed on the model. Any exceptions that are thrown in the execution of this method will cancel the transaction.
- The save after commit event is dispatched after a successful commit. This event is sent with the newly saved object. This event occurs after the commit and any exceptions thrown during the execution of the event will not affect the transaction.

Further reading:

- `\Magento\Framework\Model\ResourceModel\Db\AbstractDb`

Describe how to extend existing entities. What mechanisms are available to extend existing classes, for example by adding a new attribute, a new field in the database, or a new related entity?

- You can substitute the class for another class that extends the original class. This can be risky if you get into a rewrite substitution.
- Use the magic getters and setters (not good for testing) or `getData()` and `setData()` (not as “pretty”).
- Create a new class that is related to the original class. With this, create a new table that has a one-to-one relation to the original class and data structure.

Make sure that the new table entities have been created.

Describe how to filter, sort, and specify the selected values for collections and repositories. How do you select a subset of records from the database?

Collections

- Filter: `$collection->addFieldToFilter()`
- Sort: `$collection->addOrder()`
- Choose column: `$collection->addFieldToSelect()`
- Pagination: `$collection->setPageSize()` and `$collection->setCurPage()`

Repositories

These are a powerful feature in Magento 2. However, the simplicity of collections cannot be matched as repositories are still complex. We suggest you read through this DevDocs article to familiarize yourself with this concept.

Describe the database abstraction layer for Magento. What type of exceptions does the database layer throw?

See: `\Magento\Framework\Model\ResourceModel\Db\AbstractDb`

Exceptions:

- "Empty identifier field name" when no ID Field Name is specified.
- "Empty main table name" when no main table is specified.
- "Unique constraint violation found" when trying to insert a row with a primary key that already exists.

Additional functionality:

The resource model provides methods that make retrieving and saving rows to and from the

database table very easy.

The Zend adapter class for MySQL is: `\Zend_Db_Adapter_Pdo_Mysql`.

Describe the EAV load and save process and differences from the flat table load and save process.

Source: `\Magento\Eav\Model\Entity\AbstractEntity`

The interface for loading and saving EAV entities is identical to simple models. The process for saving and loading the initial row is similar. Once the initial entity row is loaded for a flat table, the load operation is complete. EAV entities require loading attributes from the entity attribute tables. The same is true for saving.

See `_loadModelAttributes()` in the above file for details about the loading process. Magento creates a UNION select to load attributes from each of the entity type tables to locate the applicable attribute values.

See `_collectSaveData()` in the above file for how the saving process works.

What happens when an EAV entity has too many attributes? How does the number of websites/stores affect the EAV load/save process? How would you customize the load and save process for an EAV entity in the situations described here?

If too many attributes are created, the website will begin to slow down. That is because of the large select statements that are needed to request the attribute values. It is recommended to enable flat tables (for products and categories). However, these flat tables are limited by the number of columns available.

Depending on the MySQL version, the maximum number of columns is fixed at 4096 (MySql Docs - Column Count Limit). This is less for prior versions of MySQL (StackExchange - MySql Column Limit on Table).

You can limit the number of columns in a flat table by making attributes not “Visible in Product Listing.”

The number of websites and stores can dramatically increase the loading and saving process times. The reason for this is that every variation needs to be stored in its own row. Additionally, loading the value involves selecting at least two rows.

1.17 Demonstrate the ability to use App emulation

Points to remember:

- There are two “things” that we may need to emulate—area and a scope (website, locale, and so on).
- In order to emulate area, you use `\Magento\Framework\App\State::setAreaCode()`.
- In order to emulate a scope, you use `\Magento\Store\Model\App\Emulation`.

Area emulation

Usually you emulate an area in CLI commands or cron jobs, although there could be other cases as well. In order to emulate area, include the State object in the constructor and then set the code:

```
public function __construct(\Magento\Framework\Area\State $state) {
    $state->setAreaCode(\Magento\Framework\App\Area::AREA_FRONTEND);
}
```

Scope emulation

Use `\Magento\Store\Model\App\Emulation::startEnvironmentEmulation()` and `\Magento\Store\Model\App\Emulation::stopEnvironmentEmulation()`

See the method's signature to understand the parameters.

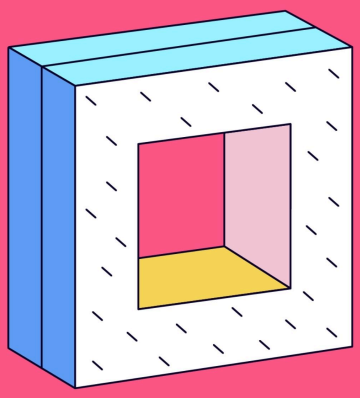
```php
public function startEnvironmentEmulation(
    $storeId,
    $area = \Magento\Framework\App\Area::AREA_FRONTEND,
    $force = false
)
```

What's the `$force` parameter do? If the current store is the same as the requested store, the `startEnvironmentEmulation` will return and not do any emulation. However, `$force` forces the emulation of the same store to happen.

Also refer to the file itself: `\Magento\Store\Model\App\Emulation`

Examples of use of the scope emulation feature are:

- `module-customer/Model/EmailNotification.php`
- `module-payment/Helper/Data.php`
- `module-sales/Model/Order/Pdf/Invoice.php`



Objective 2

Working with Databases and EAV

10% of the test / 7 questions

Adobe Commerce Expert Developer Prep Guide, AD0-E716

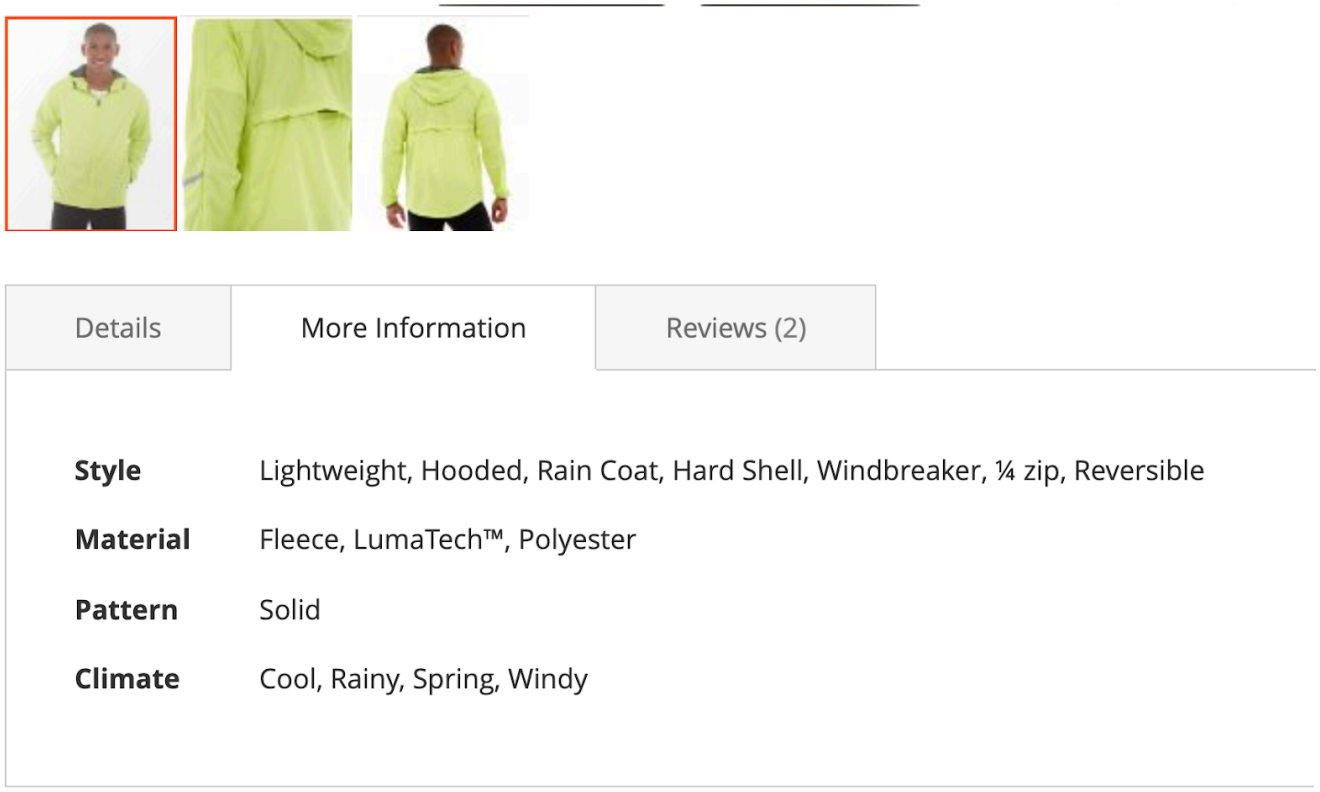
SwiftOtter

2.01 Manipulate EAV attributes and attribute sets programmatically

Describe EAV attributes, including the frontend/source/backend structure. How would you add dropdown/multiselect attributes?

Frontend: formats or adjusts the value of the attribute on the frontend. The value of the attribute's frontend_model property must be set to a class that implements `Magento\Eav\Model\Entity\Attribute\Frontend\FrontendInterface` (or extends `Magento\Eav\Model\Entity\Attribute\Frontend\AbstractFrontend`, which is more meaningful). The key method to implement is `getValue()` which takes an entity model as a parameter.

The main purpose of the frontend model is to render an attribute on the storefront, on the product view page:



Source: provides a list of acceptable options for an attribute. The most basic example would

be boolean options. See the `visibility` attribute for an example.

The main purpose of a source model is to provide options for select-type attributes (select and multiselect). A source model must implement `Magento\Eav\Model\Entity\Attribute\Source\SourceInterface` or extend `Magento\Eav\Model\Entity\Attribute\Source\AbstractSource`.

Various native implementations are available, such as `Magento\Eav\Model\Entity\Attribute\Source\Config` - which allows to specify options in a config, `Magento\Eav\Model\Entity\Attribute\Source\Table` - used very often, and provides option values from the database, `Magento\Eav\Model\Entity\Attribute\Source\Boolean` - obviously provides options for boolean dropdowns.

Note, that for catalog entities, a source model may implement the `getFlatColumns()` method which is used in the indexing process, and the `addValueSortToCollection()` method which allows you to specify custom logic for sorting by this attribute. See `Magento\Catalog\Model\Product\Attribute\Source\Status` as a reference implementation

Backend: controls how the attribute's value is saved to the database. For a basic example, see `\Magento\Customer\Model\Attribute\Backend\Data\Boolean`

Backend model allows to react on load and save operations for the entity that owns an attribute. Backend model must implement `Magento\Eav\Model\Entity\Attribute\Backend\BackendInterface` or extend `Magento\Eav\Model\Entity\Attribute\Backend\AbstractBackend` which is more meaningful. Methods of interest: `afterLoad()`, `beforeSave()`, `afterSave()`, `validate()`. The latter one is an interesting example which serves the single purpose - to implement backend-level validation for attribute saving. Note that, usually you want to ensure that `AbstractBackend::validate()` is executed, since it has some valuable logic.

What other possibilities do you have when adding an attribute (to a product, for example)?

See this answer: StackExchange - Add Product Attribute Programmatically for available configuration details.

Describe how to implement the interface for attribute frontend models. What is the purpose of this interface? How can you render your attribute value on the frontend?

Example: `\Magento\Eav\Model\Entity\Attribute\Frontend\Datetime`

- Create the new frontend class that extends `\Magento\Eav\Model\Entity\Attribute\Frontend\AbstractFrontend`.
- Set the `frontend_model` for the attribute needed to customize.
- Build the needed functionality in the `getValue()` method.

The purpose for the interface is a placeholder to eventually build a service contract for frontend details.

To render a formatted attribute value on the frontend:

```
/**
*@var Magento\Catalog\Helper\Output
*/

private $productHelper;

public function getAttributeValue($attributeCode)
{
    return $this->productHelper->productAttribute(
        $this->product,
```

```
        $this->product->getCustomAttribute($attributeCode)->getValue(),
        $attributeCode
    );
}
```

Identify the purpose and describe how to implement the interface for attribute source models. For a given dropdown/multiselect attribute, how can you specify and manipulate its list of options?

Example: `\Magento\Eav\Model\Entity\Attribute\Source\Boolean`

- Extend `\Magento\Eav\Model\Entity\Attribute\Source\AbstractSource` (or implement the interface that the abstract source class implements)
- Make sure to build the `getFlatColumns()` and `addValueSortToCollection()` methods.
- Return results in the `getAllOptions()` method.

If you need to manipulate an existing source, create an `after` plugin for the `getAllOptions()` method.

Identify the purpose and describe how to implement the interface for attribute backend models. How (and why) would you create a backend model for an attribute?

Example: `\Magento\Catalog\Model\Product\Attribute\Backend\Boolean`

- Create a new class that implements `\Magento\Eav\Model\Entity\Attribute\Backend\BackendInterface` or extends `\Magento\Eav\Model\Entity\Attribute\Backend\AbstractBackend` (easier)

Describe how to create and customize attributes. How would you add a new

attribute to the product, category, or customer entities? What is the difference between adding a new attribute and modifying an existing one?

Because attribute information is data-related and not schema-related, use the `Setup/InstallData.php` or `Setup/UpgradeData.php` classes. Inject an instance of `\Magento\Eav\Setup\EavSetupFactory` to create these attributes.

A few classes extend `\Magento\Eav\Setup\EavSetupFactory` that provide additional features for specific entity types:

- Categories: `\Magento\Catalog\Setup\CategorySetup`
- Customers: `\Magento\Customer\Setup\CustomerSetup`
- Orders, Invoices, Shipments, and Credit memos: `\Magento\Sales\Setup\SalesSetup`
- Quotes: `\Magento\Quote\Setup\QuoteSetup`

To add a new attribute, see this class: `\Magento\Customer\Setup\Patch\Data\AddCustomerUpdatedAtAttribute`.

Adding a new attribute uses the `addAttribute()` method in the `EavSetup` class. Modifying an attribute uses the `updateAttribute()` method in the `EavSetup` class. When you modify an attribute, you can pass an array of settings to update on that attribute.

2.02 Demonstrate the ability to extend the database schema

Describe the install/upgrade workflow. Where are setup scripts located, and how are they executed?

Upgrade and install scripts are located in a module's `Setup` directory.

Schema (tables/columns) is installed first, then data fixtures. While the old `InstallSchema`-style classes still work (see here), these classes are deprecated.

The source for the `bin/magento setup:upgrade` command is found in `\Magento\Setup\Console\Command\UpgradeCommand`.

The CLI command then calls `setup/src/Magento/Setup/Model/Installer.php::installSchema()` which calls `handleDBSchemaData()`. This function is good to review to learn how Magento handles the upgrade scripts.

Which types of functionality correspond to each type of setup script?

- Schema: database tables, columns, keys, and foreign keys.
- Recurring: triggered after every time that `setup:upgrade` is run whether or not an install or upgrade happened.
- Data: fields and row in a table.

Demonstrate use of schema.

Declarative schema places the structure of the database into XML. This provides the benefit of making upgrades easier in that the instructions for the upgrade come from one source. Before schema, the install/upgrade scripts were very clunky and error-prone. It was sometimes difficult to determine what the final table's structure should be as that could be determined through multiple versions of upgrades.

These new and incredible schemas are found in your module's `etc/db_schema.xml` configuration file. Ultimately, their use is simple—once you get the code base upgraded.

Example:

```
<schema>
    <table name="price_list">
        <column xsi:type="int" name="id" unsigned="true" nullable="false" identity="tr
        <column xsi:type="varchar" name="code" length="10" comment="Price List Code"/>
        <column xsi:type="datetime" name="last_sync" comment="Last Sync Time"/>
        <constraint xsi:type="primary" referenceId="PRIMARY">
```

```
            <column name="id"/>
        </constraint>
    </table>
</schema>
```

The XSD file for `db_schema.xml` files is found in: `vendor/magento/framework/Setup/Declaration/Schema/etc/schema.xsd`.

Table: this identifies the overall table that is created. The `name` attribute is required and unique. This determines the name of the table in the database. Because it is unique, this is the element which is used for merging table configuration across multiple `db_schema.xml` files. As such, you can add a table into a Magento table like:

```
<!-- This adds a 'delivery_date' column into the Magento 'quote' table -->
<table name="quote">
    <column xsi:type="datetime" name="delivery_date" nullable="false" comment="Deliver
</table>
```

To add a table to the database, specify its configuration in `db_schema.xml`. To remove a table, remove the table from where it is declared in `db_schema.xml`. Obviously, you shouldn't modify core files to remove a table.

One common problem which will leave you with an error is if a module is disabled that contains the original/core declaration for a table but another module depends on the disabled module.

Column: this configures the column to be added to the database. Each column must have a:

- `name`: this does what you think it will do
- `xsi:type`: the type of column, for example, `boolean`, `date`, `int`, `text`, `varchar`

You can specify other attributes:

- `default` : determines the column's default value in MySQL
- `disabled` : removes the column from the table
- `unsigned` : positive and negative or just positive numbers?
- `padding` : the size of an integer column

To rename a column, use the `onCreate="migrateDataFrom(entity_id)"` attribute. This copies data from the old to the new (see `\Magento\Framework\Setup\SchemaListener::changeColumn()`). Also, you must update your module's `db_schema_whitelist.json` to include both the old and the new columns.

Constraint: this specifies a primary, unique, or foreign key. There are two types of constraints in schema: internal and foreign.

Internal constraints reference columns in the table. For example, to create a primary key:

```
<constraint xsi:type="primary" referenceId="PRIMARY">
    <column name="id"/>
</constraint>
```

Foreign keys do not utilize the `column` tag, but rather specify all their details as attributes in the `constraint` tag.

Further reading:

- DB Schema

How to manipulate columns and keys using declarative schema?

You manipulate this information by adding `column`'s and `constraint`'s to a module's `etc/db_schema.xml` file. Then, you run `bin/magento setup:db-schema:upgrade`

Further reading:

- DB Schema

What is the purpose of whitelisting?

DB Schema creates and deletes database tables and columns. If this schema controlled the entire database, other tables (such as the inclusion of Wordpress in the same database) would be deleted, as they are not found in the DB Schema.

As such, DB Schema only operates on (ie removes) tables and columns for which it has been explicitly allowed (whitelisted), and in tables that are defined in the schema (`etc/db_schema.xml`).

To generate a whitelist, run:

```
bin/magento setup:db-declaration:generate-whitelist
```

To reiterate: if you have whitelisted tables or columns in a module and then you disable the module and run `setup:upgrade`, these tables or columns will be deleted. This can be a fatal mistake.

If desired, you can run the whitelist generation for a specific module thanks to the `--module-name` attribute.

2.03 Demonstrate the ability to import / export data from Adobe Commerce

Points to remember:

- You can import/export products, customers, addresses
- You may need to update PHP settings (like php.ini)
- Magento first validates all records the imports, this can be time-consuming
- Products import is probably the most important one, in Magento you can import all product types, categories and images

General import flow

1. Prepare csv file
2. Configure import behavior
3. Upload the file and start validation process
4. Proceed with the import itself

Here is an example of the import files format: Magento 2 Import/Export Sample Files

Further reading:

- Data Import

Products import overview

Standard product import follows the general steps described above. However, there are some difficulties related to products which are:

- **How to import custom options?** In the same csv file a `custom_options` field in the special format, like:

  ```
  name=Custom Yoga
  Option,type=drop_down,required=0,price=10.0000,price_type=fixed,sku=,option_title=
  option>
  ```

- **How to import custom attributes?** In the same way as regular, just add another column.
- **How to import different product types and their configurations?** We cover configurable products separately, for other product types see the reference example above.
- **How to import extension attributes?** This requires customization, since Magento does not know how to process extension attributes.
- **How to import values for different stores?** This can be done by adding multiple rows, one per product-store pair.

Importing Configurable Products

Use `product_variations` column to specify information about connection configurable-to-simple. Specify each simple in its own line. See the reference CSV file for example.

Images import

Images have to be uploaded to the `var/import/images`, then create a special csv file with sku, image labels and image files as shown in this example: Importing Product Images

Import categories

Categories are imported within the product file. See the reference for example. This allows the creation of categories and category-to-product association at the same time. You have to specify the valid category path in order to import hierarchy.

Custom import

You can extend import/export functionality by developing an import for your custom entity. See the guide here: Custom Import Entity

At the same time, sometimes native import (especially for products/categories) does not meet the requirements. First of all, performance requirements. Keep in mind that Magento runs a strict validation process which takes a lot of time, and can be a problem itself.

In the event that you decide to implement the import yourself, the best way to achieve optimal performance is to import data directly into tables. This is a very challenging task by itself, but it allows the achievement of maximal possible performance with products import.

You can also adjust or upload a new example import file. These are added in `Magento\ImportExport\Model\Import\SampleFileProvider`. See this for an example.

2.04 Describe how to use patches and recurring set ups to

modify the database

How to use Data and Schema patches?

Patches run incremental updates against the database. They perform operations that are not possible to do in the XML declaration.

Once a patch is applied, the patch is stored in the `patch_list` table and never run again.

Data patches must be in your module's `Setup/Patch/Data` directory and must implement the `\Magento\Framework\Setup\Patch\DataPatchInterface` interface. Schema patches are found in the `Setup/Patch/Schema` directory and must implement the `\Magento\Framework\Setup\Patch\SchemaPatchInterface`.

There are 3 methods that must be implemented for these interfaces:

- `getAliases`: if this patch ever changes names, this returns other names for the patch.
- `apply`: takes action.
- `getDependencies` (static): this returns an array of patches that this patch is dependent on. In other words, this patch class will run after those specified in the `getDependencies` output.

Additionally, if you wish to make this patchable to be rolled back, you can implement the `\Magento\Framework\Setup\Patch\PatchRevertableInterface` interface. This interface specifies a `revert()` method so you can take action when the module is being uninstalled.

Finally, if you wish to convert your upgrade scripts to DB Schema and need to ensure that the patch was only run once, then you can utilize the `\Magento\Framework\Setup\Patch\PatchVersionInterface` interface. Here, you specify the `getVersion()` method which allows you to associate the patch to a specific version. Magento's goal is to get away from version numbers being associated with database upgrades, instead of relying on patches and the more intuitive DB Schema to get the job done.

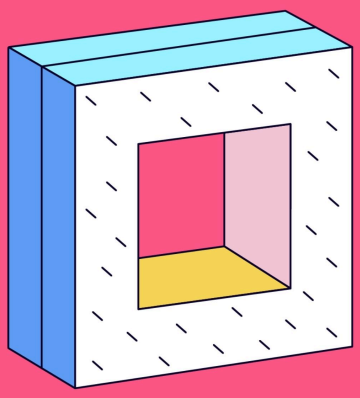
Further reading:

- Develop data and schema patches

How to manage dependencies between patch files?

When creating a patch, you need to provide implementations for three methods. One of them is the static `getDependencies()` method:

```
public static function getDependencies()
{
    return [
        \MyCompany\MyModule\Setup\Patch\Schema\FirstPatch::class
    ];
}
```



Objective 3

Developing with Admin

6% of the test / 4 questions

Adobe Commerce Expert Developer Prep Guide, AD0-E716

SwiftOtter

3.01 Demonstrate the ability to update and create grids and forms

Define form structure, form templates, grids, grid containers, and elements. What steps are needed to display a grid or form?

UI Components comprise a complex system spanning backend configuration and frontend functionality. They are assembled with declarative XML that allows deeply nested components. The configuration is output as JSON on the page which is then used to bootstrap a network of Javascript modules. These are small, reusable pieces of dynamic functionality which allow for building complex user interfaces.

This section will not go into great detail on how to build a UI component. It will provide you with sample files that you need to study. The best way to learn is to actually implement a UI component. They appear scary on the surface but after some trial and error, they become more approachable.

Forms

Forms are a list of labels and inputs laid out in a vertical format. For a form to work, you must provide it the following ingredients: a data source and fieldset/field configuration. The most simple form component configuration in Magento is: `vendor/magento/module-cms/view/adminhtml/ui_component/cms_block_form.xml`

As you review this field's configuration, you will see that while it is somewhat verbose, Magento gives tremendous flexibility in this customization.

Grids

In UI Component speak, grids are listings. These represent data stored in a tabular format.

The simplest version of a listing UI component is: `vendor/magento/module-cms/view/adminhtml/ui_component/cms_block_listing.xml`

Implementing a grid or form UI component:

When you build functionality, if you can find where Magento already has something similar, you just saved yourself a lot of time. Above is listed two UI components that will provide the basic foundation for getting started.

- Copy the component into `app/code/AcmeWidgets/ProductPromoter/view/adminhtml/ui_component`.
- Rename it to something that is applicable for you.
- Search the document for all references to the previous UI component (and all derivatives of that) and replace with the comparable updated string.
- Add this UI component to your layout XML. Example: `vendor/magento/module-cms/view/adminhtml/layout/cms_block_edit.xml`
- Find all references to Magento PHP classes within the UI component and build comparable classes within your module.

Describe the grid and form workflow. How is data provided to the grid or form? How can this be process be customized or extended?

Grid

Magento automatically wires up the data and inserts it through the `mui/index/render` controller action. This action loads data from the CMS data provider: `\Magento\Cms\Ui\Component\DataProvider`

You can alternatively configure the grid to fetch and return the data. See an example here: `vendor/magento/module-cms/etc/di.xml`, looking at the `<type name="Magento\Framework\View\Element\UiComponent\DataProvider\CollectionFactory">` node.

Form

Using the `dataSource` nodes, Magento needs to know what `dataProvider` to retrieve data from. Example: `\Magento\Cms\Model\Page\DataProvider`.

Customizing the process

Plugins are ideal as they allow you to modify the output from the data provider methods (`getData()`). You can also create a preference for an entirely different data provider.

Further reading:

- DevDocs: Overview of UI Components
- Alan Storm: Introducing UI Components
- Magento StackExchange: Debugging UI Components

3.02 Extend Grid actions

There are three things that can be considered as "grid actions":

- Row-based action (like Edit, Delete) is the most popular action. When you click on the row, the row-based action may be triggered.
- Mass actions, these are the ones that are listed in the dropdown at the top left corner. You should select a list of rows using checkboxes in the first column and then apply a mass action.
- Inline editing. Some grids (not all!) support inline editing. An example is the CMS grid.

Row-based actions.

You have to create a special Actions class which implements possible row-based actions. Usually they are just links. You can customize that Actions column, but this is a challenging task.

Note, that row-based actions usually do not work well with the inline editor.

Here is an example of adding "Edit" action in `Magento_Catalog/view/adminhtml/`

`ui_component/product_listing.xml` under the `<columns>` node:

```
<actionsColumn name="actions"  class="Magento\Catalog\Ui\Component\Listing\Columns\Pro
    <settings>
       <indexField>entity_id</indexField>
    </settings>
</actionsColumn>
```

And here is an example of the `prepareDataSource` method from `Magento_Catalog/Ui/Component/Listing/Columns/ProductActions.php`:

```
public function prepareDataSource(array $dataSource)
{
    if (isset($dataSource['data']['items'])) {
        $storeId = $this->context->getFilterParam('store_id');

        foreach ($dataSource['data']['items'] as &$item) {
           $item[$this->getData('name')]['edit'] = [
               'href' => $this->urlBuilder->getUrl(
                         'catalog/product/edit',
                         [
                           'id' => $item['entity_id'],
                           'store' => $storeId
                         ]
                ),
                'label' => __('Edit'),
                'hidden' => false,
            ];
        }
    }

    return $dataSource;
}
```

Mass Actions.

First thing to do when adding a support for mass actions is to add a selection column (the one with checkboxes). We will look at `Magento_Catalog/view/adminhtml/ui_component/product_listing.xml` as an example here:

```
<columns>
    <selectionsColumn name="ids" sortOrder="0">
        <settings>
            <indexField>entity_id</indexField>
        </settings>
    </selectionsColumn>
</columns>
```

Next, we should configure mass actions in the xml (same `product_listing.xml` as above).

See the example of the xml code below. Note there are 3 mass actions and they are all different - “delete” is a simple “actionable” url, “status” supports parameters (0 or 1) and “attributes” which redirects a user to a page with many fields to specify.

```
<listingToolbar>
    <massaction name="listing_massaction"
                component="Magento_Ui/js/grid/tree-massactions"
                class="\Magento\Catalog\Ui\Component\Product\MassAction">
        <action name="delete">
            <settings>
                <confirm>
                    <message translate="true">Delete selected items?</message>
                    <title translate="true">Delete items</title>
                </confirm>
                <url path="catalog/product/massDelete"/>
                <type>delete</type>
```

```
                <label translate="true">Delete</label>
            </settings>
        </action>
        <action name="status">
            <settings>
                <type>status</type>
                <label translate="true">Change status</label>
                <actions>
                    <action name="0">
                        <type>enable</type>
                        <label translate="true">Enable</label>
                        <url path="catalog/product/massStatus">
                            <param name="status">1</param>
                        </url>
                    </action>
                    <action name="1">
                        <type>disable</type>
                        <label translate="true">Disable</label>
                        <url path="catalog/product/massStatus">
                            <param name="status">2</param>
                        </url>
                    </action>
                </actions>
            </settings>
        </action>
        <action name="attributes">
            <settings>
                <url path="catalog/product_action_attribute/edit"/>
                <type>attributes</type>
                <label translate="true">Update attributes</label>
            </settings>
        </action>
    </massaction>
</listingToolbar>
```

See more details (in particular callback feature) here: UI Mass Actions

Finally, `Magento_Catalog` implements `MassAction` php class. The only purpose of this class is to verify the actions are allowed for a given user, see `\Magento\Catalog\Ui\Component\Product\MassAction` for the reference.

Inline Editing.

We will be using the `Magento_Cms/view/adminhtml/ui_component/cms_page_listing.xml` UiComponent's config in this example.

First we should enable the editor itself. It is done in the columns/settings node:

```
<editorConfig>
    <param name="clientConfig" xsi:type="array">
        <item name="saveUrl" xsi:type="url" path="cms/page/inlineEdit"/>
        <item name="validateBeforeSave" xsi:type="boolean">false</item>
    </param>
    <param name="indexField" xsi:type="string">page_id</param>
    <param name="enabled" xsi:type="boolean">true</param>
    <param name="selectProvider" xsi:type="string">cms_page_listing.cms_page_listing.c
</editorConfig>
```

After that for each editable column we specify its editor configuration:

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

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

And of course we should implement the editing action (in this case `cms/page/inlineEdit`).

3.03 Demonstrate the ability to create modifier classes

Things to remember:

- Modifiers are used to dynamically (on the PHP level) change the configuration of a listing/form component
- Modifiers may change both configuration and data
- Modifiers are generic listing/form feature, but in order to enable them for a custom entity one has to extend Magento\Ui\DataProvider\ModifierPoolDataProvider class in the custom DataProvider

Modifiers for products.

They are defined in the `Magento/Catalog/etc/adminhtml/di.xml`:

```
<virtualType name="Magento\Catalog\Ui\DataProvider\Product\Listing\Modifier\Pool" type
    <arguments>
        <argument name="modifiers" xsi:type="array">
            <item name="attributes" xsi:type="array">
                <item name="class" xsi:type="string">Magento\Catalog\Ui\DataProvider\P
                <item name="sortOrder" xsi:type="number">10</item>
            </item>
            <item name="priceAttributes" xsi:type="array">
                <item name="class" xsi:type="string">Magento\Catalog\Ui\DataProvider\P
                <item name="sortOrder" xsi:type="number">10</item>
```

```
            </item>
        </argument>
    </arguments>
</virtualType>
```

You can define your modifier in the same fashion.

Next there is a modifier class (we look at the `Magento\Catalog\Ui\DataProvider\Product\Modifier\Attributes` class).

First, it has to extend `Magento\Ui\DataProvider\Modifier\ModifierInterface` and implement two methods:

```
interface ModifierInterface
{
    public function modifyData(array $data);

    public function modifyMeta(array $meta);
}
```

Here is an example of the `modifyData()` method implementation:

```
public function modifyData(array $data)
{
    if (!empty($data) && !empty($this->escapeAttributes)) {
        foreach ($data['items'] as &$item) {
            foreach ($this->escapeAttributes as $escapeAttribute) {
                if (isset($item[$escapeAttribute])) {
                    $item[$escapeAttribute] = $this->escaper->escapeHtml($item[$escape
                }
            }
```

```
        }
    }

    return $data;
}
```

You can also check Catalog Product's form modifiers to see more complex examples.

3.04 Demonstrate the ability to restrict access to ACL

Describe how to set up a menu item and permissions. How would you add a new menu item in a given tab? How would you add a new tab in the Admin menu? How do menu items relate to ACL permissions?

Menu items are configured in the `etc/adminhtml/menu.xml` XML configuration file.

Here is a very basic example:

```
<config>
    <menu>
        <add id="AcmeWidgets_ProductPromoter::PromotionBuilder"
            title="Promotion Builder"
            translate="title"
            module="AcmeWidgets_ProductPromoter"
            sortOrder="50"
            parent="Magento_Catalog::inventory"
            action="promotion/builder"
            resource="AcmeWidgets_ProductPromoter::PromotionBuilder"
            />
    </menu>
</config>
```

The primary node here is `<add/>`. The following will discuss the attributes listed above.

- `id`: this value is used when creating a hierarchy of menu items. The `parent` module refers to another `<add/>` node's `id` parameter.
- `title`: the title of the menu item
- `translate`: which parameters to translate. This is usually just `title`.
- `module`: the module that is associated with the menu item
- `sortOrder`: determines the order in which this item appears
- `parent`: menu item that will host the current menu item. If no `parent` is specified, this item will appear on the sidebar.
- `action`: the controller action to send the click of the menu item. If no `action` is specified, this item will act as a header.
- `resource`: the ID of the ACL entry to validate the user's permissions.

The other nodes that are available are `update` and `remove`. Both take the `id` attribute relating to another menu item's `id` attribute.

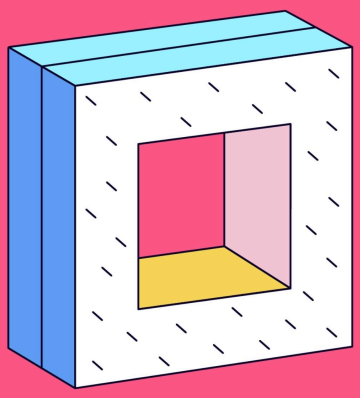
Describe how to check for permissions in the permissions management tree structures. How would you add a new user with a given set of permissions? How can you do that programmatically?

Every item in the ACL has an ID. This ID is used as the lookup field for determining whether or not the action is allowed.

The tree structure is useful for allowing or disallowing a particular group. As a result, that particular resource ID might not have an entry in the database. Magento does a “depth-first search” (see: `\Zend_Acl`) starting at the deepest node (the one with the ID you are looking up) and works up until it finds an Allow or Deny for the current user.

To create a user with a particular set of permissions (role), create the user and specify its role.

Programmatically, you could obtain an instance of `\Magento\User\Model\UserFactory`, set the details for the use, and specify `role_id` for this instance (`$user->setData(‘role_id', $roleId);`).



Objective 4

Customizing the Catalog

6% of the test / 4 questions

Adobe Commerce Expert Developer Prep Guide, AD0-E716

SwiftOtter

##4.01 Apply changes to existing product types and create new ones

Identify/describe standard product types (simple, configurable, bundled, etc.). How would you obtain a product of a specific type?

Magento provides the capability for various product types. The default products are:

- `simple` : represents the basic unit of inventory on the shelf.
- `configurable` : provides a select list of options based on chosen attributes.
- `grouped` : customer selects how many they want to order of each product in a list.
- `bundled` : a list of multiple configuration options. `virtual` : an intangible product or digital goods.
- `downloadable` : virtual product that can be downloaded.

Retrieving product(s) of a specific type:

```
namespace AmceWidgets\ProductPromoter\Model;

class ProductLoader
{
    private $productRepository;
    private $searchCriteriaBuilder;

    public function __construct(
        \Magento\Catalog\Api\ProductRepositoryInterface $productRepository,
        \Magento\Framework\Api\SearchCriteriaBuilder $searchCriteriaBuilder
    ) {
        $this->productRepository = $productRepository;
        $this->searchCriteriaBuilder = $searchCriteriaBuilder;
    }

    public function getBundleProducts()
```

```
    {
        $searchCriteria = $this->searchCriteriaBuilder
            ->addFilter('type_id', \Magento\Bundle\Model\Product\Type::TYPE_CODE)
            ->create();

        return $this->productRepository->getList($searchCriteria)->getItems();
    }
}
```

What tools (in general) does a product type model provide?

The product type model:

- is responsible for handling data as it is passed to and from the database.
- handles loading child products (if applicable).
- loads and configures product options.
- checks whether the item is saleable.
- prepares the product to be added to the cart.

What additional functionality is available for each of the different product types?

- Simple product represents the basic unit of inventory.
- Configurable products load the list of their children and the attribute associated with them.
- Downloadable products are virtual at heart, but after the sale allows the user to download content. It provides the ability to limit the download to the logged in user.
- Grouped products are similar to configurable in that it stores a list of children, but no attributes are used in loading the children.
- Bundled products contain functionality to load the bundle options, configure the final products, and display the price.

4.02 Modify and extend existing Catalog entities

There are two major Catalog entities that you may want to modify and extend - `Product` and `Category`. `Category` is covered below.

In terms of modifying and extending Product entity, typical operations are:

- Customize product types (see objective 4.01)
- Operate with attributes (EAV or extension attributes), see objectives: 1.02, 2.01, 6.01
- Modify major functional areas: Pricing & Indexing (see objective 4.03), Stock (see objective 4.04
- Work with various products relations such as cross-sells, upsells, linked products (see Related Products, Up-Sells and Cross-Sells)
- Work with static content (images, videos, custom pdf-files related to products), partially covered in objectives 2.01, 2.03. Customizations in this area are usually relatively difficult and are out of scope for this exam (we should verify this).

One particular customization technique that worth separate mentioning is the use of Custom Options. See user guide for details of their functionality: Customizable Options.

The main benefit of custom options is that they are automatically indexed and taken into account in all possible price calculation processes, plus they are very flexible. So we can use custom options to implement many different non-standard requirements, especially those that require modification of the price calculation logic, which is very intricate and sometimes bewildering.

Custom options could be alternatives to configurable products in some situations. The main “rule of thumb” to consider a custom option is - inventory. Custom options do not possess any inventory and this is rather difficult to change. So in case our requirements have something to do with inventory, then custom options may not be the best solution. Otherwise it is always useful to think if unusual custom requirements can somehow be implemented with custom options.

They are not very well documented in general, check here: Customizable Option Interface

And refer to the source code of the `Product` and custom options subframework in the Catalog module. See: `Magento\Catalog\Model\Product`, `vendor/magento/module-catalog/Model/Product/Option/*`

Categories

Points to remember:

- Root category is used to group categories within a store. It is never shown to the customer.
- `parent_id = 0` is the identifier for the grandparent of root categories. ID `0` does not exist in the database and is used as a placeholder.
- Category name is the only required category attribute that is not automatically specified.

Describe category properties and features. How do you create and manage categories?

Categories are easily created in the Magento Admin panel under the Catalog menu item. Categories represent a tree-view hierarchy in that there is one parent category and everything descends from that parent.

You can also create categories with code using the `\Magento\Catalog\Api\Data\CategoryInterfaceFactory` object and saving with `\Magento\Catalog\Api\CategoryRepositoryInterface`.

Categories are an EAV-type and are stored in the `catalog_category_entity` table.

Describe the category hierarchy tree structure implementation (the internal structure inside the database).

The hierarchy is stored in the `path` column in the `catalog_category_entity` table. The first number in the path is the root category; the last number is the `entity_id` of the current row. By exploding the `path` column by the `/` character, you can determine the path to the current category. Additionally, the `level` column gives insight as to how deep this category is within the tree.

Please note that the root category is never shown to the frontend user. This is used to group categories within a store.

Further reading:

- `\Magento\Catalog\Model\Category\Tree`

What is the meaning of parent_id 0?

Parent ID `0` serves as the grandparent to all root categories (ID `0` only exists in code and not in the database). Magento automatically creates the category with ID `1`, which is the parent to all root categories.

`Category::ROOT_CATEGORY_ID = 0;` is specified in the `Category` model but is not referenced or used anywhere else.

`Category::TREE_ROOT_ID = 1;` is specified in the `Category` model and is used extensively in determining the tree path in the `path` column.

Here is a screenshot of the usage in the database:

row_id	entity_id	created_in	updated_in	attribute_set_id	parent_id	created_at	updated_at	path	position	level	children_count
1	1	1	2147483647	3	0	2016-12-30 18:42:23	2018-03-06 13:30:17	1	0	0	107
2	2	1	2147483647	3	1	2016-12-30 18:42:23	2017-09-07 17:41:55	1/2	1	1	105
3	3	1	2147483647	3	2	2016-12-30 18:42:49	2018-01-24 18:11:05	1/2/3	4	2	4

How are paths constructed?

Paths are constructed by taking the parent ID of the current category and ascending up the tree until the root category's parent (ID: 1) is reached. This is then imploded separated by a `/`.

A good way to see how paths are constructed is by inspecting the code that constructs the paths: `\Magento\Catalog\Model\ResourceModel\Category::changeParent()`

Which attribute values are required to display a new category in the store?

- Category name
- Available Product Listing Sort By (defaults to Use All)
- Default Product Listing Sort By (defaults to Use Config Settings)

What kind of strategies can you suggest for organizing products into categories?

The most important aspect of this is how a user wants to browse the website. Performing user studies and analysis is better than a theoretical strategy that can be contrived and then presented.

Beyond that, products should be organized into logical groups. For example, this might be “Computers” or “Smartphones” or “Carrying cases.” Under “Computers” would likely be “Laptops,” “Desktops” and “All-in-One.”

Content managers can use the sort order column to show more popular or strategic products first.

4.03 Demonstrate the ability to manage Indexes and customize price output

Indexing

Indexes are Magento “innovation” that is aimed to improve performance where native MySQL functionality (plus data structure imposed by Magento) is not sufficient. The typical example - prices, stock levels and so on.

It is important to understand when indexing is really necessary. The brightest example is a category page where we should display layered navigation, provide sort by price functionality and so on.

For example, price calculation is very complex, so it is nearly impossible to sort by price if calculating a price on-the-fly. For this purpose Magento duplicates all the price calculation logic in SQL and stores prices into a special table that is used for sorting/filtering.

Please note, that indexes usually have website scope.

Further reading:

- Indexing Overview

Prices

Points to remember:

- Base price, special pricing, and catalog rules apply to the price visible on a product page.
- Tiered pricing, options price, tax / VAT (depending on the point in the checkout process), and shopping cart rules determine the price in the shopping cart.

Identify the basic concepts of price generation in Magento. How would you

identify what is composing the final price of the product? How can you customize the price calculation process?

Magento offers many layers of pricing calculation in the application. Here are the primary calculations that take place for the price shown on a product page: (`\Magento\Catalog\Model\Product\Type\Price::calculatePrice()`)

- Base price (`price` attribute) or existing price
- Special pricing
- Catalog rules

Once a quantity has been determined for a product (ie. added to the cart), several other options apply:

- Tiered pricing (applicable to quantity, customer group, and website)
- Options price
- Tax / VAT

When determining a product's final price (what is shown on the product's page), Magento works through each one of these.

`\Magento\Catalog\Model\Product\Type\Price` is where these calculations take place. Customization can happen with plugins (`afterCalculatePrice` , for example) or replacing the entire price calculation class.

Describe how price is rendered in Magento. How would you render price in a given place on the page, and how would you modify how the price is rendered?

Pricing renderers are setup in `vendor/magento/module-catalog/view/base/layout/default.xml` . In this file, a block is created with the name `product.price.render.default` . You can use this block to render pricing elsewhere in the application (example: `vendor/magento/module-downloadable/view/frontend/layout/`

`catalog_product_view_type_downloadable.xml`).

The templates for these renderers are found here: `vendor/magento/module-catalog/view/base/templates/product/` .

Additionally, there is a JS UI component to display prices (Price Rendering). These templates can be found in `vendor/magento/module-catalog/view/base/web/template/` .

Catalog Price Rules

Points to remember:

- Catalog rules are indexed and the data is stored in `catalogrule_product_price` .

Identify how to implement catalog price rules. When would you use catalog price rules? How do they impact performance? How would you debug problems with catalog price rules?

Setting up catalog price rules is done in the admin panel under Marketing > Catalog Price Rule. Here, the content manager can easily filter for product(s) to apply this rule to, allow only specific customer groups to utilize it, and apply discounts. This is not to be confused with Shopping Cart Rules.

Catalog price rules are a great way to set up sales on a more global basis than special pricing allows for. A product's special price is easy to set up for a one-off product or simple group of products. Catalog price rules can be used to apply a discount to a set of (or all) products. Unlike special pricing, catalog price rules can apply to certain customer groups.

Catalog price rules will slightly affect performance. These rules are not indexed by the price indexer. They are indexed, however, by the Catalog Product Rule index and the applicable rule price resides in the `catalogrule_product_price` table.

Debugging rules

Debugging rules for a merchant is much easier if you have access to a copy of their production database (we have built a tool that we use to always keep secure copies of production data with no custom information: Driver).

Catalog rules are indexed. This means that debugging has to isolate the problem in two places:

- is the problem the data going into the indexed table?
- is the problem the data coming out of the indexed table and not being applied to the product?

The catalog rule indexes are built in: `\Magento\CatalogRule\Model\Indexer\IndexBuilder`

4.04 Explain how multi-source inventory impacts stock (program level)

Things to remember:

- MSI disables original inventory functionality
- There is no locking during checkout
- Stock management is done via the reservation mechanism. Each reservation has to be compensated in order to count a stock level accurately

Overall functionality organization.

The MSI functionality is broken into many modules (about 66). These modules are broken into features. For example, for “sales” feature there are modules:

- `module-inventory-sales`
- `module-inventory-sales-admin-ui`

- `module-inventory-sales-api`
- `module-inventory-sales-frontend-ui`

Usually there is an “api” module which only has interface, a “regular” module which includes implementation and other modules like `admin-ui`, `graph-ql` and so on.

Most important modules are:

- `module-inventory-sales`
- `module-inventory-shipping`
- `module-inventory-reservation`
- `module-source-selection`
- `module-inventory-indexer`
- `module-shipping`

For example, `module-inventory-sales` disables original `cataloginventory` functionality (Magento_InventorySales/etc/events.xml):

```
<!--There is no need to register product sale and reindex stock items, as in multi sou
<event name="checkout_submit_all_after">
        <observer name="inventory" instance="Magento\CatalogInventory\Observer\Checkou
</event>
<event name="sales_model_service_quote_submit_before">
        <observer name="inventory" instance="Magento\CatalogInventory\Observer\Subtrac
</event>
```

Note that the naming convention is sometimes confusing. For example there are modules - `module-inventory-sales` and `module-sales-inventory`.

The reservation mechanism

The best way to understand the reservation mechanism is to look into the database, in the `inventory_reservation` table.

You will see the records like:

```
reservation_id : 19
stock_id       : 1
sku            : some-sku
quantity       : -4.0000
metadata       : {"event_type":"order_placed","object_type":"order","object_id":"24"}
```

and

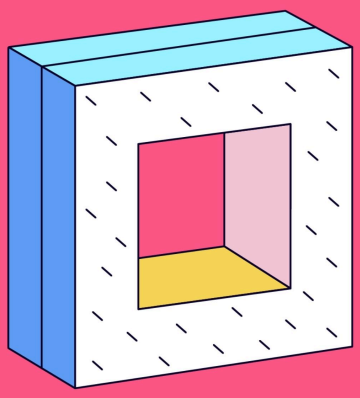
```
reservation_id : 207
stock_id       : 1
sku            : another-sku
quantity       : 2.0000
metadata       : {"event_type":"invoice_created","object_type":"order","object_id":"11
1 row in set (0.00 sec)
```

So, every time something that causes and inventory change happens, Magento issues a record in the reservation table. Finally all reservations have to cancel each other out. Then the indexer will pick up that information and update an index.

Sometimes reservations do not cancel each other. In this case there are some CLI commands to fix inconsistencies. See more here: Reservations & Inventory CLI Reference

Further reading:

- Inventory Management Overview



Objective 5

Customizing Sales Operations

6% of the test / 4 questions

Adobe Commerce Expert Developer Prep Guide, AD0-E716

SwiftOtter

5.01 Demonstrate the ability to develop new payment methods or customize existing payment methods

Points to remember:

- Payment methods implement `\Magento\Quote\Api\Data\PaymentMethodInterface` and `\Magento\Payment\Model\MethodInterface`
- Magento has three types of payment methods: offline, gateway and hosted.

Offline Payment Methods

Offline methods are the simplest ones. They are located in `vendor/magento/module-offline-payments/`. Typically there is a configuration in `etc/config.xml` and `etc/payment.xml`, see `vendor/magento/module-offline-payments/etc/config.xml` and `vendor/magento/module-offline-payments/etc/payment.xml`

There is a `MethodInstance` class which extends `Magento\Payment\Model\Method\AbstractMethod` and implements a series of methods.

The functionality and configuration of such method's behavior is done via implementing methods from `Magento\Payment\Model\Method\AbstractMethod` and also by defining a set of `MethodInstance`'s protected variables.

Gateway Methods

Gateway methods perform a remote call on checkout to process a payment. While it is possible to follow the same approach as in “Offline Payment Methods”, i.e. implementing the whole logic in one `MethodInstance` class extending `Magento\Payment\Model\Method\AbstractMethod` Magento has a special framework for gateway-type payment methods. It is relatively well documented here: Payments Integrations

Please note, that as of 2.4 Magento removed Braintree (and Authorize) from its core repository,

so there are no good references out-of-the-box anymore.

Overview

- `paypal/module-braintree-core` is a good example of gateway payment methods.
- The payment method is defined in `etc/payment.xml`.
- The module is specified in `config/default/payment/braintree/model` in `etc/config.xml`. This aligns with the declarations in `etc/di.xml`.
- A virtual type reference in `etc/config.xml` extends `Magento\Payment\Model\Method\Adapter`
- Values are passed into the adapter like `code`, `formBlockType` (the payment renderer for the admin order form), `infoBlockType` (the payment renderer for view order page). Then, there are the complex types:
    - `commandPool`: hosts a list of commands to execute as necessary. Note that before a command is executed, a value handler is checked to see if this command can be executed. For example, for the `void` command, `can_void` is sent to the value handler. Here is an example command: `\PayPal\Braintree\Gateway\Command\GatewayCommand`.
    - `validatorPool`: validators extend `\Magento\Payment\Gateway\Validator\CountryValidator`.
    - `handlerPool`: value handlers return configuration values. There should always be a `default` (`\Magento\Payment\Gateway\Config\ValueHandlerPool::DEFAULT_HANDLER` handler. Here is Braintree's default configuration handler: `\PayPal\Braintree\Gateway\Config\Config`

Hosted

Hosted method is something like PayPal Express Checkout (when a customer is redirected from the website to PayPal's website, logs in there, fulfills all the payment information on the paypal's side and then redirected back by PayPal to Magento).

There is not such a detailed framework for this type compared to gateways, so you should implement the whole functionality yourself.

Describe how to troubleshoot payment methods.

Similar to the above methods for troubleshooting shipping methods. For simpler payment methods, check to ensure that the method's `isAvailable()` is returning `true`. This is originally declared in `\Magento\Payment\Model\Method\AbstractMethod`.

More complex implementations such as Braintree, PayPal, and Authorize.net are more difficult to troubleshoot. To enable better security, these solutions have transitioned to iframes or a JavaScript token solution. The IPN (instant payment notification), the route where PayPal sends updates about the payment status, is not easily debugged.

The good news is that many of these payment methods have a `Debug` configuration setting. Turning this on enables verbose logging in `var/log` and can help point to where problems are occurring.

5.02 Demonstrate the ability to add and customize shipping methods

Points to remember:

- Shipping methods implement `\Magento\Shipping\Model\Carrier\CarrierInterface`

Describe shipping methods architecture. How can you create a new shipping

method?

Shipping methods are configured with XML in the `etc/` folder.

Creating a new shipping method:

- Create a new group in `etc/adminhtml/system.xml` for `carriers/[shipping_code]`.
- Add `etc/config.xml` and create the path for `default/carriers[shipping_code]` with default values (if necessary) for what was specified in `system.xml`.
- Add a `<model/>` node to point to the class that contains the shipping method.
- Create a new class that implements `\Magento\Shipping\Model\Carrier\CarrierInterface` and possibly extends `\Magento\Shipping\Model\Carrier\AbstractCarrierOnline`.

What is the relationship between carriers and rates?

Carriers provide a list of available rates. For example, for UPS, the available rates might be Overnight, Overnight AM, and Ground.

Describe how to troubleshoot shipping methods and rate results.

- Find the shipping method's class.
- Set a breakpoint in `\Magento\Shipping\Model\Carrier\AbstractCarrierOnline::canCollectRates()` to make sure that the carrier is enabled. Remember that plugins can change the returned value.
- Set a breakpoint in the implementation for `\Magento\Shipping\Model\Carrier\AbstractCarrierInterface::collectRates()` and step through the request formulation and the response parsing.

Where do shipping rates come from?

Shipping rates can come from an API, calculations, or be fixed.

Examples:

- API: `\Magento\Fedex\Model\Carrier`
- Calculations: `\Magento\OfflineShipping\Model\Carrier\Tablerate`
- Fixed: `\Magento\OfflineShipping\Model\Carrier\Flatrate`

How can you debug the wrong shipping rate being returned?

See the answer above for "Describe how to troubleshoot shipping methods and rate results."

Further reading:

- Add Custom Shipping Carrier

5.03 Demonstrate the ability to customize sales operations

Demonstrate ability to customize sales operations

Points to remember:

- Going into Magento admin > Stores > Order Status > Assign Order Status to State allows you to change the status associated with the default order state.
- There are two types of credit memos (as associated with the payment method): online and offline.

Describe how to modify order processing and integrate it with a third-party ERP system.

There are several readily accessible touchpoints to integrate Magento orders with an external system.

REST API

Magento 2 includes an extensive API that provides access to orders. Specifically, the sales order repository has been made available through the API: `/rest/V1/orders`. The Swagger documentation provides information on how to filter for a specific subset of orders.

The use case for this would be an external service that queries Magento for the latest orders. This service operates as a pull operation, retrieving orders from Magento, and pushing into the ERP.

Events

Magento provides a number of events that indicate when an order is complete. For example, `sales_order_place_after` is triggered when the order has been placed.

One thing to keep in mind with events is the length of time to process. If the code that is executed is long-running, your checkout will appear very slow. At a minimum, you should use a cron job to separate the long-running processes from the web process. You can also use RabbitMQ (available in Magento Open Source as a module).

Describe how to modify order processing flow. How would you add new states and statuses for an order? How do you change the behavior of existing states and statuses?

By default, an order is placed, then invoiced, then shipped (marking the order as complete).

There are a number of ways to change this. First, invoices can be automatically created. An event such as `sales_order_place_after` is an ideal time for this. Hooking into a 3rd-party system that will automatically generate orders also could modify this flow.

Adding new states and statuses for an order.

You can create new order statuses in the admin panel, Stores > Order Statuses. This can be also updated in the database.

Clicking on the Assign Status to State button in this panel allows you to change the order status associated with the default state.

States are hardwired into the `\Magento\Sales\Model\Order` class. While you can create plugins to hook into and change these states, unintended consequences might arise.

Described how to customize invoices. How would you customize invoice generation, capturing, and management?

Invoices are created when payment has been captured. There are, however, a number of cases where payment capture doesn't matter. An excellent example is a terms payment method (where a customer is invoiced at the end of the month) when an ERP is involved. In these cases, the order needs to be marked as complete because the ERP handles invoicing at the end of the month. Automatically creating the invoice is necessary here.

Invoices are built against the `\Magento\Sales\Api\Data\InvoiceInterface` service contract. Additionally, invoices are extendable, so you can create extension attributes for them.

Invoices are also accessible through the Magento REST API.

Describe refund functionality in Magento. Which refund types are available, and how are they used?

Refunds (credit memos) in Magento are used to return money back to the purchaser. Often this is a result of the customer returning part of (or all of) their order.

There are two refund types: online and offline. In the admin order view, you will see a button called “Credit Memo.” Creating a credit memo here will generate an offline credit memo. This means that if an online payment method was used (Braintree, Paypal, Authorize.net), the payment provider will not be contacted to issue the refund: it happens offline. This is a point of confusion for many merchants.

An online credit memo (when the external payment's provider is contacted to create the refund) is created from an invoice. To create an online memo, go to an order, click on the

Invoices tab, choose an invoice, and then click Credit Memo. Please note that this option is only available if the payment method is online.

Demonstrate ability to use quote, quote item, address, and shopping cart rules in checkout. Describe how to modify these models and effectively use them in customizations.

Each of the following entities extends `\Magento\Framework\Model\AbstractExtensibleModel` so you can use extensible attributes as discussed in Chapter 4.

Quote

Quotes are the “storage container” for a customer's shopping cart session. They contain details about what is being purchased, the current totals, and customer information. A quote is associated with the current session in `\Magento\Checkout\Model\Session::getQuote()`.

Quotes are stored in the `quote` table.

This is a frequent place to make customizations as this is integral to the customer order experience. The model containing the quote's data is: `\Magento\Quote\Model\Quote`

Quote Item

Quote items contain the contents of a visitor's shopping cart. They link the quote (shopping cart) to products. These items are stored in `quote_item`. This table is a good place to store additional information about the item (such as custom information about taxes).

A quote item is represented by `\Magento\Quote\Model\Quote\Item`.

Address

Every quote has at least a billing address and a shipping address. A quote address is represented by `\Magento\Quote\Model\Quote\Address`. Quote addresses also store a list of items that are related to this address.

Shopping cart rules

Shopping cart rules apply discounts to items in the cart. The `Magento\SalesRule` module handles this logic.

Describe how to customize the process of adding a product to the cart. Which different scenarios should you take into account?

There are a number of ways to add items to the cart in Magento 2:

- Frontend, through Magento application
- Backend, through Magento application
- From the wishlist
- Reordering a product
- During quotes merge (visitor has items in their cart, logs in, the current quote is merged with the quote associated with the logged in customer)
- REST API

If customization needs to happen when adding items to the cart, it is important to not hook into a specific controller, but rather use plugins on functions that are used in every situation.

Describe add-to-cart logic in different scenarios. What is the difference in adding a product to the cart from the product page, from the wishlist, by clicking Reorder, and during quotes merge?

Product page: `\Magento\Checkout\Controller\Cart\Add`

Events triggered (object provided):

- `checkout_cart_add_product_complete`
- `checkout_cart_product_add_after` (in cart entity, applicable to frontend, backend, and REST)
- `sales_quote_product_add_after` (in quote entity)

- `sales_quote_add_item` (in quote entity)

A buy request is created with the incoming quantity and other product options. This is passed into the cart model which interacts with the quote model to generate items to add.

Wishlist page: `\Magento\Wishlist\Controller\Index\Cart`

Events triggered:

- `checkout_cart_product_add_after` (in cart entity, applicable to frontend, backend, and REST)
- `sales_quote_product_add_after` (in quote entity)
- `sales_quote_add_item` (in quote entity)

The buy request is used to check that it still is correct. The wishlist item is then told to add itself to the cart. This calls the cart `addProduct` method which is what the above path calls.

Reorder: `\Magento\Sales\Controller\AbstractController\Reorder`

Events triggered:

- `checkout_cart_product_add_after` (in cart entity, applicable to frontend, backend, and REST)
- `sales_quote_product_add_after` (in quote entity)
- `sales_quote_add_item` (in quote entity)

The reorder controller calls the cart's `addOrderItem` method. This loads the buy request and calls the cart's `addProduct` method.

Merge: `\Magento\Quote\Model\Quote::merge()`

Events triggered:

- `sales_quote_add_item` (in quote entity)

Describe the difference in behavior of different product types in the shopping cart. How are configurable and bundle products rendered? How can you create a custom shopping cart renderer?

Simple: appear as one line item in the shopping cart.

Configurable: appear as one line item in the cart. The parent product's title is shown with the chosen option visible.

Bundle: appear as one line item in the cart with all the options displayed below the title.

Grouped: appear as multiple line items: one for each item chosen.

Rendering configurable and bundle products

These products are rendered as specified in `view/frontend/layout/checkout_cart_item_renderers.xml` (example: `vendor/magento/module-bundle/view/frontend/layout/checkout_cart_item_renderers.xml`). The `renderer` block specified the `as` parameter to determine which child to render for the given input.

Describe the available shopping cart operations. Which operations are available to the customer on the cart page?

- Change item quantity
- Delete item
- Edit item
- Move to wishlist (if enabled)
- Add gift message (if enabled)
- Update cart (after changing item quantities)
- Apply / remove coupon
- Go to checkout

How can you customize cart edit functionality?

Clicking `edit` on a product returns the visitor to what looks like the product page, except that the URL starts with `checkout/cart/configure`.

The configuration controller is in `\Magento\Checkout\Controller\Cart\Configure`. This action simply loads the quote item and sends it to the product helper to render.

How would you create an extension that deletes one item if another was deleted?

Create an event observer for `sales_quote_remove_item` or a plugin for the `\Magento\Quote\Model\Quote::removeItem()` method.

How do you add a field to the shipping address?

- Add a new column to the `sales_order_address` table.
- Create a `view/frontend/layout/checkout_index_index.xml` file.
- Replicate the path to `<item name="shipping-address-fieldset" xsi:type="array">` that is found in:

  ```
  vendor/magento/module-checkout/view/frontend/layout/
  checkout_index_index.xml
  ```
- Add the `children` node and specify the child node.

5.04 Explain how to customize totals

Describe how to modify the price calculation process in the shopping cart. How can you add a custom totals model or modify existing totals models?

To create a custom total modal, you will need to wire it up in `etc/sales.xml` and create the model that will handle the processing.

`vendor/magento/module-tax/etc/sales.xml` provides an example of how the tax module links this up:

```
<config>
    <section name="quote">
        <group name="totals">
            <item name="tax" instance="Magento\Tax\Model\Sales\Total\Quote\Tax" sort_o
                <renderer name="adminhtml" instance="Magento\Sales\Block\Adminhtml\Ord
                <renderer name="frontend" instance="Magento\Tax\Block\Checkout\Tax"/>
            </item>
        </group>
    </section>
</config>
```

Every total model extends `\Magento\Quote\Model\Quote\Address\Total\AbstractTotal`.

Demonstrate ability to customize the shopping cart

Points to remember:

- The only method that is common means of adding an item to the cart is `\Magento\Sales\Model\Quote::addItem`.
- Only one shopping cart rule as applied with a coupon code can be used at a time. Multiple rules can be applied at once.

Describe how to implement shopping cart rules.

Shopping cart rules are primarily built through the admin panel (the other option is adding it programmatically through `Setup/UpgradeData.php`). This is done in Marketing > Cart Price Rules.

What is the difference between cart price rules and catalog rules?

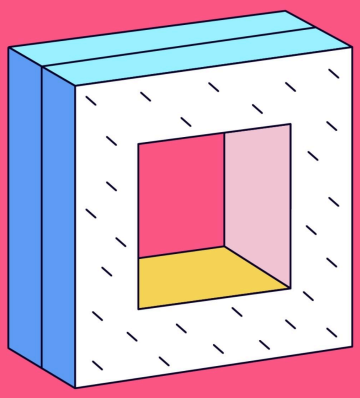
Cart price rules apply to products that are in the cart. Catalog rules apply to products before being added to the cart.

As a result of being added to the cart, sales rules have more options for customizing as they can offer free shipping, discounts on the entire cart, and specific coupon codes.

How do sales rules affect performance? What are the limitations of the native sales rules engine?

Sales rules affect performance negatively in the shopping cart and order creation because processing the rules can be a significant task (especially if there are complex filtering conditions). Performance will slow down the more global (not limited to a customer group or website) the rule, and the more rules that are available, and the fewer that require a coupon code.

Limitations: only one coupon code can be used at a time. You cannot have one product in your cart and another appear for free (or a discount). Rules cannot add other products to the cart. Rules only apply to the item they were designated to work with.



Objective 6

APIs and Services

4% of the test / 3 questions

Adobe Commerce Expert Developer Prep Guide, AD0-E716

SwiftOtter

6.01 Demonstrate the ability to create new APIs or extend existing APIs

Points to remember:

- Magento API based on interfaces located in the module's `Api`, and `Api/Data` folders
- `Api` interfaces are functional and `Api/Data` interfaces only describe data
- Implementation of Api interfaces governed by preferences in `di.xml`
- Interfaces could be automatically exposed to the WebApi via `etc/webapi.xml`
- WebApi calls are executed in their own area (`webapi_soap`, `webapi_rest`)

Magento APIs

Magento API interfaces located in Module's Api and Api/Data folders. Generally speaking these interfaces do not have to extend anything. Some interfaces are typical, for example Repository - is an interface, whose implementation performs CRUD operations. The fact that it doesn't extend anything is rather unfortunate.

Data interfaces define an entity's properties.

See `Magento\Catalog\Api\Data\CostInterface` for a simple example.

Note:

- Every method must have a `return` docblock.
- `array` is not a valid return type. Instead, specify what this is an array of: `string[]`, `Magento\Catalog\Api\Data\ProductInterface[]`.

WebAPI

Each (functional) Api interface can be exposed via WebAPI. For doing so, one must configure

Api endpoints in the `etc/webapi.xml`.

Here is an example from `Magento/Catalog/etc/webapi.xml`:

```
<?xml version="1.0"?>

<routes>
    <!-- Product Service -->
    <route url="/V1/products" method="POST">
            <service class="Magento\Catalog\Api\ProductRepositoryInterface" method="sa
            <resources>
            <resource ref="Magento_Catalog::products" /> <!-- specifying authorization
            </resources>
    </route>
    <route url="/V1/products/:sku" method="PUT">
            <service class="Magento\Catalog\Api\ProductRepositoryInterface" method="sa
            <resources>
            <resource ref="Magento_Catalog::products" />
            </resources>
    </route>
</routes>
```

Note the key parts:

- Service node - defines an interface and method
- Resources/resource - defines an ACL resource (the same as the standard Magento Admin ACL)

There are multiple ways to authenticate with Magento WebApi. See more details here.

Note that there are role-based, `anonymous` and `self` authentications. `Self` allows a customer to retrieve their shopping cart and MyAccount information. It is important that Magento uses REST API in the checkout.

See more details about WebAPI here.

In order to perform a remote call to Magento WebApi you may need to send a parameter which could be an object. Usually these objects are either SearchCriteria objects or Data Api classes. Since both have very strict definitions of their fields you can (and must) use JSON to represent these objects.

See this for an example.

API necessities

There are a couple of required components to ensure that Magento can correctly assess the details from the API.

- Every method must have a `@return` declaration. Because PHP does not have List types, your `return` type might be `array`, but the `@return` declaration must declare what *type* of return, like: `string[]` or `\Magento\Catalog\Api\Data\ProductInterface[]`.
- There must be a `@param` specified for incoming parameters. Declared in the code is not good enough.
- **Code smell:** if you *ever* return a `Json` result from a controller—you just made a terrible smell. Use the API instead.
- If you need to return a key-value array, build this out in an interface and model. It's not that hard.

```
interface ProductRepositoryInterface
{
    /**
     * Create product
     *
     * @param \Magento\Catalog\Api\Data\ProductInterface $product
```

```
     * @param bool $saveOptions
     * @return \Magento\Catalog\Api\Data\ProductInterface
     * @throws \Magento\Framework\Exception\InputException
     * @throws \Magento\Framework\Exception\StateException
     * @throws \Magento\Framework\Exception\CouldNotSaveException
     */
    public function save(
        \Magento\Catalog\Api\Data\ProductInterface $product,
        $saveOptions = false
    );
}
```

Extension Attributes and API

See Objective 1.02, for further details. Note, that one has to extend `DataInterface` from `\Magento\Framework\Api\ExtensibleDataInterface` and its implementation from `\Magento\Framework\Model\AbstractExtensibleModel`.

The latter brings in the functionality to create/fetch `ExtensionAttributes` object which wraps all extension attributes (objects/values).

Extension attributes can also be filtered by permissions, like this:

```
<?xml version="1.0"?>
<config>
    <extension_attributes for="Magento\AsynchronousOperations\Api\Data\OperationInterf
        <attribute code="start_time" type="string">
            <resources>
                <!-- see this here -->
                <resource ref="Magento_Logging::system_magento_logging_bulk_operations
            </resources>
        </attribute>
```

```
    </extension_attributes>
</config>
```

Extension Attributes vs Custom Attributes

One question that often causes confusion - what is the difference between Extension Attributes and Custom Attributes in terms of Magento Api?

Custom Attributes used to represent custom EAV attributes and usually they are part of Customer and Product entities.

The reason why we need yet another "attributes" in the WebApi protocol definition. It is static (the definition) while attributes may be added by a developer. So, the protocol allows `custom_attributes` which is an array of arbitrary objects.

Usually you don't have to do anything with Custom Attributes, because the most common case to encounter them is in the WebAPI payload. And since, the most common entity for custom EAV attributes is Product whose Data Api class is the same as model, you can access custom EAV attributes via direct getters and setters.

The situation, however, may differ for Customer and Customer Address entities. In this case Data class and model are different classes, so you may need to fetch CustomAttributes object from the data object to get a value of a custom EAV attribute.

6.02 Demonstrate the ability to use the queuing system

Things to remember:

- Magento uses RabbitMQ as a message broker
- Magento can be both a publisher and a consumer
- In order to consume messages Magento uses cron job, which may cause issues (messages may get lost, the job may take too long to get executed, previous job may

affect the execution of consumer and so on).

Configuration

There are four configuration files all in `<module>/etc/`:

- `communication.xml` - defines topics, handlers, schemas. In this file you define what is the format of requests and responses (whether they implement an interface or a generic strings), which classes will process messages.
- `queue_consumer.xml` - defines a configuration for a consumer. You can specify which class consumes messages of which queue.
- `queue_topology.xml` - the file in which you declare queues
- `queue_publisher.xml` - the file in which you configure publisher classes, specify type of connection and so on.

Note:

- If you are building functionality that changes configuration, and if existing queue consumers need to be updated, you can call the `put` method on the `PoisonPillPutInterface`.

See documentation for more details.

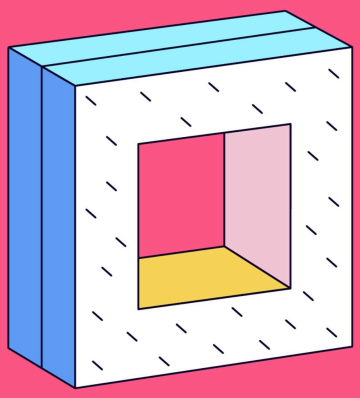
Create a publisher and a consumer

Implementation of a publisher and consumer is not strictly regulated by Magento. However, there are few classes and interfaces that could be useful.

- `Magento\Framework\MessageQueue\PublisherInterface` - an interface that defines the “publish” method.
- `Magento\Framework\MessageQueue\Publisher` - the class you typically use to actually publish a message.

Note that there are similar `Consumer` counterparts, but you don't have to use them.

Another note - the whole Message Queue framework is a separate component which is located at `vendor/magento/framework-message-queue`. Also look at `vendor/magento/module-message-queue`.



Objective 7

Adobe Commerce Cloud architecture

16% of the test / 11 questions

Adobe Commerce Expert Developer Prep Guide, AD0-E716

SwiftOtter

7.01: Demonstrate knowledge of Adobe Commerce architecture/environment workflow

Platform-as-a-Service Architecture

Platform-as-a-Service (PaaS) environments in your Cloud project are deployed as a set of virtual service containers. We'll discuss the available services in more detail in future sections, but the key services deployed into each container are the web server (NGINX with PHP-FPM) and the MySQL/MariaDB database server. Other available services for each environment are Redis, OpenSearch and RabbitMQ.

For the Starter plan, *all* environments are PaaS environments, while in Pro this applies only to integration environments.

Changes to your project code and file-based configuration are intended to be deployed through Git-based merges up through your environment hierarchy (e.g., Integration to Staging to Master/Production). We'll be covering the details of Git, environment hierarchy, and branching/deployment throughout the course.

While a PaaS environment corresponds with a Git branch, not all branches correspond with environments! Your Cloud repository can contain any number of "inactive" Git branches, but "active" branches are those with a deployed Paas environment.

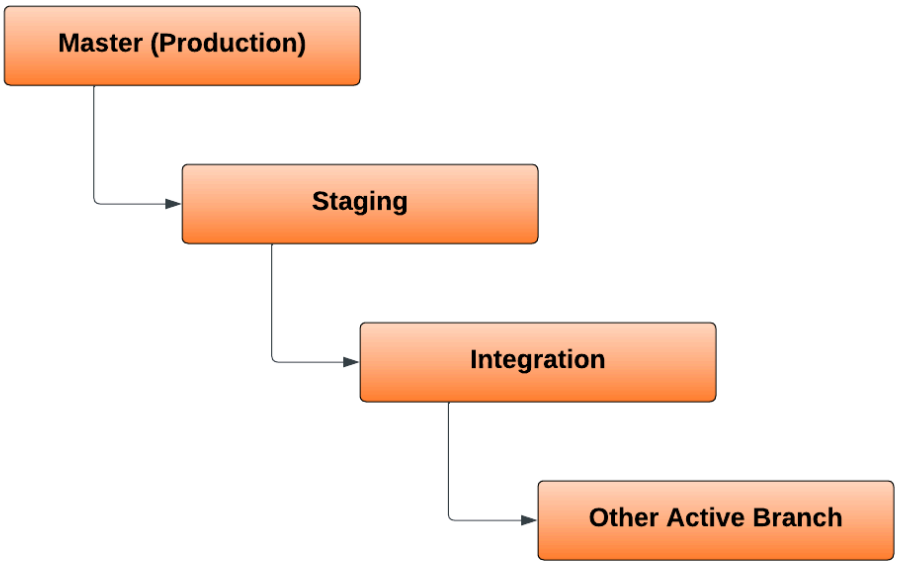
There are differences between Starter and Pro in terms of the limits on active PaaS environments and what environments are initially provisioned.

Starter

Again, all environments in a Starter plan consist of PaaS containers, including the production environment. When a Starter project is provisioned, there will be *one* environment - Master - which serves as the production site. ("Master" and "Production" are synonymous on Starter).

Three additional active environments can be created on Starter. While it's up to you to create

these environments and their hierarchy, Adobe strongly recommends immediately creating a Staging environment as a child of Master, and then an Integration environment as a child of Staging. One additional active environment can then be created. By following these recommendations, you'll end up with an environment hierarchy like this:



You can, of course, choose to branch your fourth active environment from Staging rather than from Integration, depending on what development and deployment workflow you intend to use.

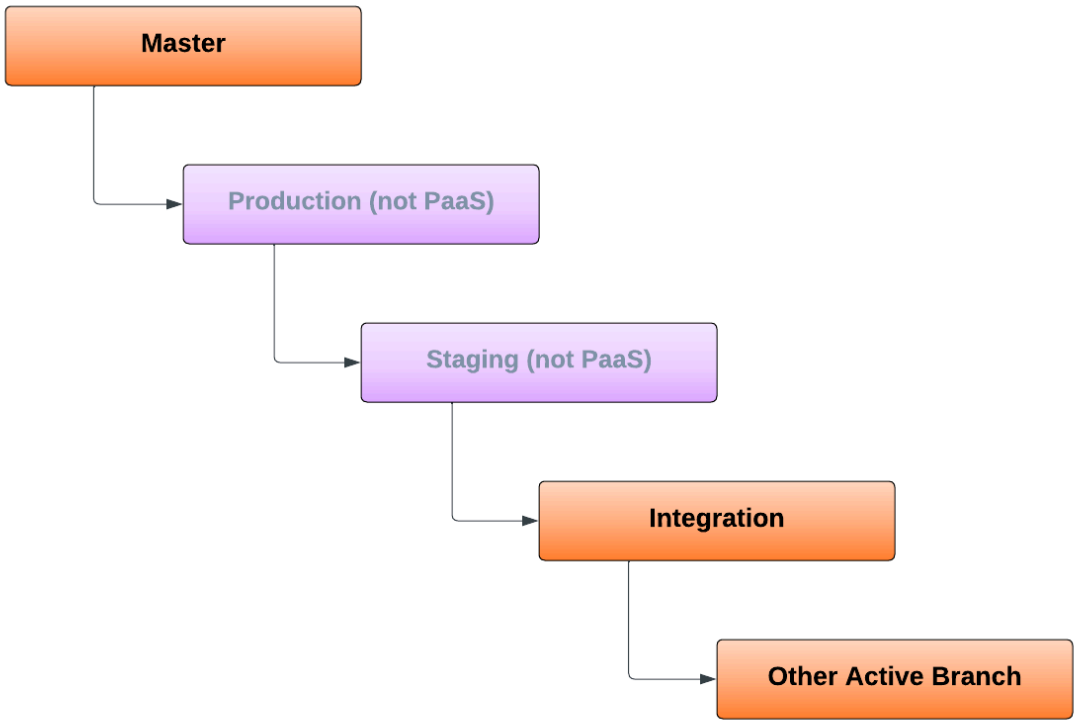
Pro

The Production and Staging environments in a Pro plan are provisioned on different architecture (which we'll cover next), but your project will *also* be provisioned with an Integration PaaS environment. The Master environment, which sits at the top of the hierarchy above Production, is also PaaS-based. (The Master environment is intended to be a mirror of Production code state, for debugging site issues without disrupting the Production environment.)

One additional active PaaS environment can be created on Pro.

While Production and Staging are not PaaS environments, they are still Git-based and have a hierarchical relationship with the other environments.

Presuming you create an additional active environment branched from Integration, your Pro hierarchy will look like this (only Master, Integration, and "Other Active Branch" are PaaS environments):



Enhanced Integration Environments

It is possible to open a support ticket to request a PaaS environment be upgraded to an "Enhanced Integration Environment," which increases the power and performance. The official documentation seems to note that this applies only to older Cloud projects and that projects provisioned today already use these up-sized environments, but it's important to know what an Enhanced Integration Environment is, in case a question on the topic shows up in your certification exam.

You can read more in the knowledge base article.

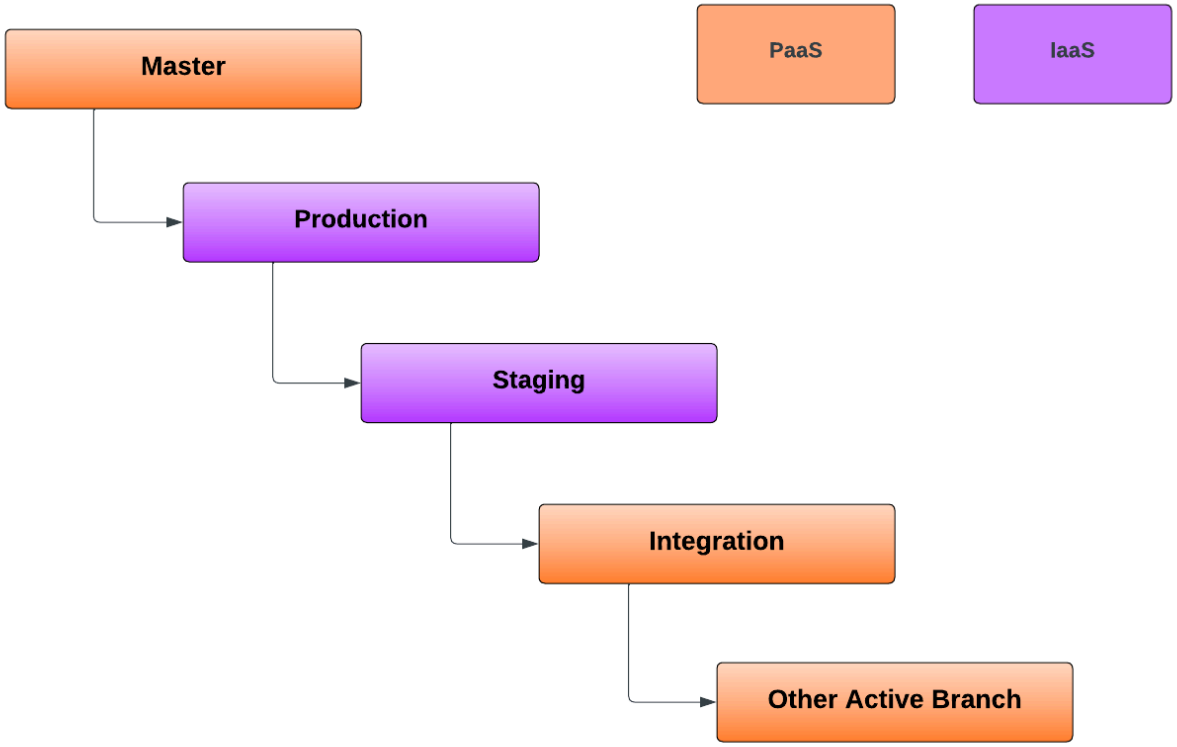
Further Reading

- Starter Architecture Summary
- Pro Architecture Summary

Infrastructure-as-a-Service Architecture

Unlike the PaaS environments, the Staging and Production environments on a Pro project are provisioned on dedicated servers in what Adobe refers to as Infrastructure-as-a-Service (IaaS) infrastructure.

You can choose either Amazon Web Services (AWS) or Microsoft Azure for your IaaS hosting.



IaaS environments are *not* involved on a Starter project.

While Pro Staging and Production environments are based on Git branches from the repository hierarchy, new environments cannot be branched from them, only from the Integration environment.

The Staging and Production environments use identical architecture, including an Elastic load balancer in front of *three* fully redundant and highly available web nodes, each with a static IP address.

Each node (or zone) is a virtual machine with its own instances of the following:

- Web server (NGINX and PHP-FPM)
- Redis (one active and two replicas)

The following services are shared between all zones:

- File mounts for key directories (using GlusterFS)
- OpenSearch
- Galera database cluster, with one MariaDB database per node

> **Core Concept**
>
> Because the IaaS environments apply only for Staging and Production in Pro projects, there are a few differences between these two environments and all others, in terms of configuration, log access, and file locations. We will point these out where they apply, by specifically noting the distinctions for the Pro Staging and Production environments.

Scaling

Scaled architecture is available on Pro accounts with the Pro48 cluster or greater. We've already mentioned the default three web nodes that are part of Pro architecture (the web tier); with scaled architecture, there are also 3 nodes for database and other services like OpenSearch and Redis, which are referred to as the service tier.

The service tier can be scaled vertically, meaning increasing power and available memory.

The web tier can be scaled both vertically and horizontally. This means that in addition to up-sizing the individual web nodes, additional nodes can be created to spread traffic out further.

Further Reading

- Pro Architecture Summary
- Scaling

7.02: Demonstrate a working knowledge of cloud project files, permission, and structure

The Filesystem

Notable Files and Directories

The file structure and core components of Adobe Commerce remain the same with a Cloud build, but there are a few notable additions.

The following files are Cloud-specific configuration files that we will delve into later:

- `.magento/routes.yaml`
- `.magento/services.yaml`
- `.magento.app.yaml`
- `.magento.env.yaml`

The following files or directories are also notable:

- `m2-hotfixes` - This directory contains patch files that are automatically applied in deployments.
- `php.ini` - PHP configuration values here will be automatically applied in the Cloud environments.

- `magento-vars.php` - This automatically included file can be used to set config values appropriately for multi-domain routing. It's automatically applied with `auto_prepend_file` in the included `php.ini`.

The Remote Filesystem

The remote file location into which the Magento application is deployed differs depending on environment:

- Production (Pro and Starter): `/{project-id}`
- Pro Staging: `/{project-id}_stg`
- All other environments: `/app`

... where `{project-id}` is the unique ID of your Cloud project (also included in the URL of your Project Web Interface).

Code changes to the application can only be made via deployments from source control. For maximum security, the entire filesystem in the deployed environment (with a few exceptions) is read-only.

Only the following directories are writeable in the Cloud environments:

- `var`
- `pub/static`
- `pub/media`
- `app/etc`
- `/tmp`

The contents of the project directory are rebuilt and moved into their final location when new deployments are run. To ensure file changes in writable directories persist across deployments, the above directories that exist within the project root (i.e., excluding `/tmp`) are

implemented as mounts, which can be managed with configuration in `.magento.app.yaml`. We'll explore this further in a future lesson.

7.03: Demonstrate the ability to setup multi domain based stores on Adobe Commerce Cloud (multi domain sites mix of dev work and support)

Multi-Domain Setup

When you need to serve multiple websites from a single Adobe Commerce project using multiple domains or sub-domains, configuration must be considered for multiple levels of the infrastructure and application.

Adding Domains

New domains must be added to the Cloud environment and to Fastly configuration.

In the Project Web Interface, when editing project-level settings, a Domains tab allows you to configure the domains associated with your project. You can also manage domains with the `domain:*` commands with the Cloud CLI tool.

Domains in the Fastly config can be managed from within the Magento admin, at Stores > Configuration > Advanced > System > Full Page Cache > Fastly Configuration > Domains. After configuring domains here and clicking "Activate", the config will be uploaded to Fastly.

> **Important**
>
> For Pro Staging and Production, you must open a support ticket to add new domains. The documentation is somewhat unclear about whether this applies to adding domains to the Cloud infrastructure, to the Fastly config, or both.

Store Configuration

Appropriate website or store view records to correspond with the additional domains or sub-domains must be set up in your Magento configuration. This is core Magento knowledge and outside the scope of this course.

Routes

If your separate websites are associated with entirely different domains, the route configuration matching `{all}` in `.magento/routes.yaml` should suffice to route all of them to your application.

If you're using a sub-domain, you can add new upstream routes to handle it:

```
"http://b2b.{default}/":
    type: upstream
    upstream: "mymagento:http"
```

Runtime Logic

With the above steps completed, requests for the appropriate domains will be successfully directed to your application, and Magento has multiple website or store configurations ready to serve. All that's missing is appropriate logic to load the correct website/store context for the corresponding domain.

Two concepts are in play:

- The environment variables `MAGE_RUN_TYPE` ("website" or "store") and `MAGE_RUN_CODE` are used by Magento to determine the store to load. The value of `MAGE_RUN_CODE` is matched to the code of a website or store record. (If these environment variables are not present, the default store is loaded.)
- The `magento-vars.php` file bootstrapped with your project is loaded on every request

and should contain the logic for setting the above variables based on the request hostname. Its initial state includes an `isHttpHost` function.

`magento-vars.php` should be modified according to your multi-site needs. The following is an example that will ensure a store with the code "b2b" will be loaded when the hostname is b2b.mysite.com:

```
function isHttpHost($host)
{
    if (!isset($_SERVER['HTTP_HOST'])) {
    return false;
    }
    return $_SERVER['HTTP_HOST'] ===  $host;
}

if (isHttpHost("b2b.mysite.com")) {
   $_SERVER["MAGE_RUN_CODE"] = "b2b";
   $_SERVER["MAGE_RUN_TYPE"] = "store";
}
```

Further Reading

- Set Up Multiple Websites
- Fastly Domain Management

7.04: Demonstrate a general knowledge of application services and how to manage them (YAML , PHP, MariaDB, Redis, RabbitMQ, etc)

Installed Services

Following is a summary of the services that are, or can be, installed in your Cloud environments. You are likely familiar with most of these technologies and their purposes. It's not our goal to discuss these services at length, and we'll be covering their configuration at a later point, so this is merely a bird's eye view of the main Cloud tech stack.

NGINX and PHP-FPM

NGINX serves as the web server on Cloud infrastructure, configured with PHP-FPM for the execution of PHP for any web requests.

MySQL

MySQL (or MariaDB, more accurately) is the database layer.

On the Pro Production environment, the MariaDB implementation actually includes multiple synchronous databases via Galera Cluster.

Redis

Redis is an in-memory data store. We'll discuss in a later section how Adobe Commerce uses Fastly CDN for full page caching, but Redis is the technology used for all *other* types of caching (e.g., configuration and layout update caching), as well as for session data.

The Redis CLI can be used directly when SSH'd into a Cloud environment, using the hostname provided by the service configuration (typically "redis.internal"):

```
redis-cli -h redis.internal
```

We'll be taking a closer look at both SSH access and service configuration later.

OpenSearch

OpenSearch provides high performance search capabilities for the Adobe Commerce application.

Note that OpenSearch is actually a fork of Elasticsearch, fulfilling the same API as a search implementation. Older versions of Adobe Commerce Cloud infrastructure have Elasticsearch installed and configured for search, but Elasticsearch 7.11 and later is not supported. Adobe Commerce 2.3.7-p3, 2.4.3-p2, and 2.4.4 and later support OpenSearch instead.

RabbitMQ

RabbitMQ is message broker software and can be used as the backend for Adobe Commerce's message queue framework.

RabbitMQ is an optional service in the Adobe Commerce stack; MySQL can be used to handle message queues instead. However, dedicated software like RabbitMQ provides better reliability and performance than a database implementation.

If RabbitMQ is not installed in your Cloud environments initially, it can be configured and installed as described in a later section. When the service has been added via the appropriate Cloud configuration files, Magento's `env.php` is automatically updated to configure queue settings appropriately.

Other Services

Two technologies that are not installed services in the Cloud architecture, but are nevertheless important entries in the Cloud tech stack, are SendGrid and New Relic.

SendGrid

The SendGrid SMTP proxy service is responsible for ending outbound emails from your Cloud environments. Outgoing emails can be enabled or disabled from the Project Web Interface or with the Cloud CLI tool.

Up to 12,000 outbound transactional emails per month are allowed, excluding marketing campaigns. You can request additional credits if you exceed your limit in a given month. Documentation also includes this note: "There are no hard limits on the number of emails that can be sent in Production, as long as the Sender Reputation score is over 95%."

On Pro plans only, you also have support for DomainKeys Identified Mail, which is a technology that helps prevent domain spoofing. Utilizing this domain authentication feature requires setting up specific DNS records. You can read more extensively about the process in the documentation.

New Relic

New Relic provides advanced monitoring and troubleshooting capabilities for web applications. The Cloud on-boarding process will include New Relic account activation and access.

New Relic connectivity is handled automatically in the provisioning of Pro accounts.

For Starter, the New Relic license key obtained from your account can support up to three environments. To connect an environment, the license key must be set as the value of the `php:newrelic.license` environment variable. (We'll cover the use of environment variables in a future section.) And the `.magento.app.yaml` file must be updated to make sure the New Relic extension for PHP is included:

```yaml
runtime:
    extensions:
        - newrelic
```

The available New Relic services include the following:

- **Application Performance Monitoring (APM)** is available on both **Pro and Starter**. This allows you to drill down into details and backtraces, and profiling for specific web transactions, as well as database query monitoring and more.
- **New Relic Infrastructure (NRI)** is available on **Pro Production**. This provides additional enhanced monitoring of your Amazon AWS or Microsoft Azure infrastructure.
- **New Relic Logs** is available on **Pro Staging and Production**. This provides a central dashboard aggregating log data from your entire infrastructure.

Further Reading

- SendGrid
- New Relic

Configuration

Configuration files in your Cloud application can be used to customize the services deployed into environment containers.

As a reminder, the following services are available (and most are already enabled/configured):

- MySQL
- Redis
- RabbitMQ
- Elasticsearch and OpenSearch

Configuration includes the definitions of the services to be provisioned, as well as the relationships of those services to the application (i.e., how they are made available).

> **Important**
>
> Installing or updating services in Pro Staging and Production environments cannot be done automatically with YAML configuration. A support ticket must be opened in order to update services in these environments to match the config files.

The Services Config File

A unique YAML file - `.magento/services.yaml` - is used for configuring installed services and their versions, allocated disk size for each, and other unique configuration. Here's an example of the initial configuration in `services.yaml`:

```yaml
mysql:
    type: mysql:10.4
    disk: 5120

redis:
    type: redis:6.2

opensearch:
    type: opensearch:1.2
    disk: 1024
```

Each entry has a unique ID and defines the desired service and version in the `type` property. Disk allocation can be provided (expressed in MB) with `disk`.

As an example of adding a service, the following could be added to `services.yaml` to install the RabbitMQ service:

```yaml
rabbitmq:
    type: rabbitmq:3.9
    disk: 1024
```

Relationship Config

Once a service is defined in `services.yaml`, its relationship to the application must be defined in `.magento.app.yaml`.

The `relationships` property handles this:

```yaml
relationships:
    database: "mysql:mysql"
    redis: "redis:redis"
    opensearch: "opensearch:opensearch"
```

The key used for each entry is used for the container's internal hostname, which the application can use to connect. (For example, the key "database" results in MySQL being available at host "database.internal".) In the value, the string on the left side of the colon should match the ID (key) of the service in `services.yaml`, while the right-hand string should match a particular "endpoint". (Different endpoints aren't usually configured for services, so the default value is simply the service type.)

Automatic Config in `env.php`

There are two common scenarios for services not initially provisioned in the Cloud environments:

- Installing RabbitMQ for use with the Magento messages framework.
- Installing a second Redis instance to separate session and cache storage.

It seems that configuring these services with the appropriate IDs will result in the appropriate

Magento configuration being written to `app/etc/env.php` during deployment.

For example, the RabbitMQ service definition above, once combined with an entry in `relationships` in `.magento.app.yaml`, will result in this configuration automatically being added to `env.php`:

```
  'queue' =>
  array (
    'amqp' =>
    array (
      'host' => 'rabbitmq.internal',
      'port' => 5672,
      'user' => 'guest',
      'password' => 'guest',
      'virtualhost' => '/',
    ),
    'consumers_wait_for_messages' => 0,
  ),
```

Similarly, the ID "redis-session" can be used to define a second Redis instance in `services.yaml`:

```
redis-session:
    type: redis:6.2
```

This would be the appropriate configuration in `.magento.app.yaml` for this service ID:

```
relationships:
    ...
    redis-session: "redis-session:redis"
```

With this config in place, the session configuration in `env.php` will automatically be changed to this:

```
'session' =>
array (
  'save' => 'redis',
  'redis' =>
  array (
    'host' => 'redis-session.internal',
    'port' => 6379,
    'database' => 0,
    'disable_locking' => 1,
  ),
),
```

> **Note**
>
> The `ece-tools` package will check your service versions during deployment and display notifications or warnings if the end-of-life dates for these versions are approaching or past.

Changing Service Versions

A service can be upgraded to a newer version simply by changing the `type` property appropriately in `services.yaml` and re-deploying.

Downgrading to a lower version, however, cannot be done so easily. An existing service ID cannot be changed to a lower version number; if such a downgrade is necessary, a new service instance (i.e., with a different ID in `services.yaml`) must be created. The existing service ID can simply be changed, but this is the equivalent of removing the old service and creating a new one, so be aware that this will still result in the deletion of all data! A better

option might be to leave the existing service in place and *add* another, allowing you the opportunity to move data from the old service instance once both are running.

Whether changing the service ID in `services.yaml` or adding another, make sure to update `relationships` in `.magento.app.yaml` accordingly.

Further Reading

Please note that there is additional available configuration, unique to each service type, that can be included in `services.yaml`. You should review the documentation for each service:

- Elasticsearch
- MySQL - Includes configuring multiple databases and endpoints in the same container
- OpenSearch
- RabbitMQ
- Redis

Main services documentation:

- Services Configuration
- Relationships Property

7.05: Identify how to access different types of logs

Logs

Let's delve into the topic of the various logs in your Cloud infrastructure - where logs reside, and various ways to access them.

Accessing Logs

Log information can be viewed in various ways, but the capabilities of each are not necessarily equivalent.

Method 1: Direct Access

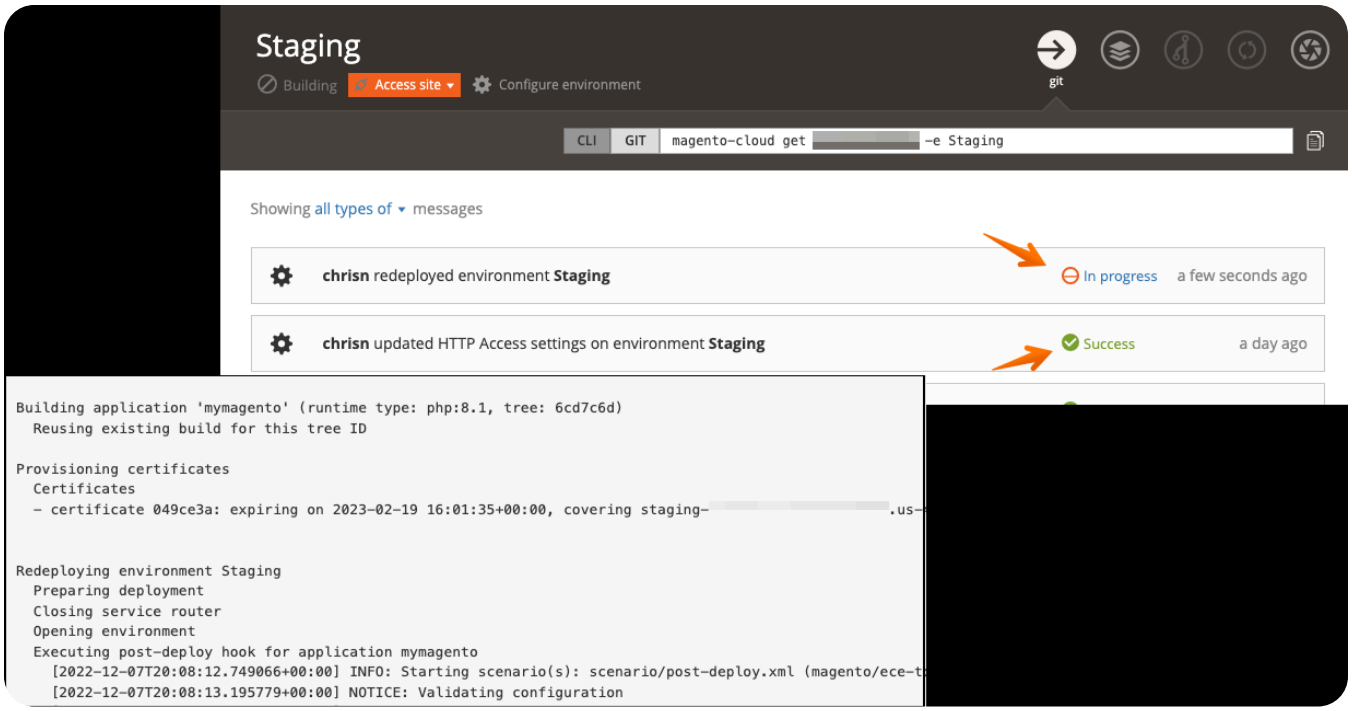
The most straightforward way of viewing logs is to connect to the appropriate environment via SSH and examine the log files directly. We'll note where each type of log is located in the remote filesystem.

Logs of *all* types (with the exception of service logs in Integration environments, as you'll see below) are available to access via this method.

> **Our Experience**
>
> Though these specific notes are not present in the Adobe Commerce Cloud documentation, working hands-on with a Cloud environment has produced these observations to be aware of:
>
> - Unlike stream output, such as in the Project Web Interface, *only* explicit logging statements are written to log files > themselves. This excludes certain messages that make stream output more readable.
> - Logging messages of *all* levels are written to file output, regardless of the minimum logging level set for stream > output.

Method 2: Project Web Interface (Stream Output)

Log information for **build and deploy operations** can be viewed in the Project Web Interface, in the messages view that displays by default when a specific environment is selected. The message status indicator can be clicked to display a dialog with the logging details. This can be done even for a deployment in progress, allowing you to monitor details throughout the operation.



What appears in this interface seems clearly to be full stream output from the operation, meaning there are details beyond log statements themselves. For example, the full output of the Composer build operation will be displayed here, but is not written to log files. The output here is the same as what is displayed in a terminal interface when performing a command that results in a Cloud deployment.

```
chrisnanninga@Chriss-MBP:02:07 PM:[redacted] $ magento-cloud environment:redeploy
Are you sure you want to redeploy the environment Staging? [y/n] Y
Waiting for the activity ayxdmak6rjeci (chrisn redeployed environment Staging):

Building application 'mymagento' (runtime type: php:8.1, tree: 6cd7c6d)
  Reusing existing build for this tree ID

Provisioning certificates
  Certificates
  - certificate 049ce3a: expiring on 2023-02-19 16:01:35+00:00, covering staging-[redacted].us-4.magentosite.cloud


Redeploying environment Staging
  Preparing deployment
  Closing service router
  Opening environment
  Executing post-deploy hook for application mymagento
    [2022-12-07T20:08:12.749066+00:00] INFO: Starting scenario(s): scenario/post-deploy.xml (magento/ece-tools version: 2002.1.13, magento/magento2-base version: 2.4.5-p1)
    [2022-12-07T20:08:13.195779+00:00] NOTICE: Validating configuration
```

If we compare what appears in the output stream to the messages written to the corresponding log files in the environment, we can see the differences:

Project Web Interface

```
Redeploying environment Staging
  Preparing deployment
  Closing service router
  Opening environment
  Executing post-deploy hook for application mymagento
    [2022-12-07T20:08:12.749066+00:00] INFO: Starting scenario(s): scenario/pos
    [2022-12-07T20:08:13.195779+00:00] NOTICE: Validating configuration
    [2022-12-07T20:08:14.448722+00:00] NOTICE: End of validation
    [2022-12-07T20:08:14.448963+00:00] INFO: Enable cron
    [2022-12-07T20:08:14.450038+00:00] INFO: Create backup of important files.
```

Log File Output

```
[2022-12-07T20:08:12.749066+00:00] INFO: Starting scenario(s): scenario/pos
[2022-12-07T20:08:13.192081+00:00] DEBUG: Running step: is-deploy-failed
[2022-12-07T20:08:13.192213+00:00] DEBUG: Step "is-deploy-failed" finished
[2022-12-07T20:08:13.195704+00:00] DEBUG: Running step: validate-config
[2022-12-07T20:08:13.195779+00:00] NOTICE: Validating configuration
[2022-12-07T20:08:13.218879+00:00] DEBUG: php ./bin/magento config:show --a
[2022-12-07T20:08:14.448722+00:00] NOTICE: End of validation
[2022-12-07T20:08:14.448834+00:00] DEBUG: Step "validate-config" finished
[2022-12-07T20:08:14.448937+00:00] DEBUG: Running step: enable-cron
[2022-12-07T20:08:14.448963+00:00] INFO: Enable cron
[2022-12-07T20:08:14.449966+00:00] DEBUG: Step "enable-cron" finished
[2022-12-07T20:08:14.450010+00:00] DEBUG: Running step: backup
[2022-12-07T20:08:14.450038+00:00] INFO: Create backup of important files.
```

The level of logging that is included in the output stream (whether in the Project Web Interface or in your terminal) is controlled by the environment variable `MIN_LOGGING_LEVEL`. (We'll discuss environment variables in more detail later.)

Method 3: Cloud CLI Command

The Cloud CLI command can be used to access and print logs from the remote environments:

```
magento-cloud log --lines 200 --tail
```

`log` is an alias of `environment:logs`. The example above includes the `--lines` option to specify a number of lines to output and the `--tail` options, which can be used to continuously tail the output of the log file.

When the command is run, an interactive prompt will allow you to choose which log you wish to view.

```
chrisnanninga@Chriss-MBP:02:33 PM:                    $ magento-cloud log
Enter a number to choose a log:
  [0] access
  [1] app-debug
  [2] app
  [3] cron
  [4] deploy
  [5] dns
  [6] error
  [7] php.access
  [8] php.debug.access
  [9] post-deploy
 >
```

The command can also be run with the log type specified:

```
magento-cloud log access
```

The logs that can be accessed via the Cloud CLI include only those outside the project root directory in the remote environment (for instance, in `/var/log` from the filesystem root).

Build and Deploy Logs

When deployments to Cloud environments occur, details are logged in the following locations:

- `var/log/cloud.log` (project directory)
  - All details are written here.
- `var/log/cloud.error.log` (project directory)
  - Error-level logs written here.
- `/var/log/deploy.log` and `post-deploy.log` (system root)
  - For PaaS environments, details specifically from the deploy and post-deploy phases are written here in the application container filesystem.
- `/var/log/platform/{project-id}_stg/deploy.log` and `post-deploy.log`
  - On the Pro plan dedicated IaaS environment, details specifically from the deploy and post-deploy phases are written here for Staging.
- `/var/log/platform/{project-id}/deploy.log` and `post-deploy.log`
  - On the Pro plan dedicated IaaS environment, details specifically from the deploy and post-deploy phases are written here for Production.

Due to the difference in location, when accessing logs for Pro Staging and Production using the Cloud CLI tool, you must specify the location (relative to `/var/log`) of the file as an argument to the command:

```
magento-cloud log platform/{project-id}_stg/deploy.log
```

In Pro Staging and Production, these log files are present only in the first web node.

Application Logs

In addition to the `deploy.log` and `post-deploy.log` files mentioned above, the root log directory in the Cloud environments contains several more log files for the web application as a whole. (By "root log directory," we mean the one outside the project root: `/var/log` in PaaS environments, `/var/log/platform/{project-id}_stg` for Pro Staging, and `/var/log/platform/{project-id}` for Pro Production.)

- `access.log` - NGINX access log
- `app-debug.log` - PHP-FPM debug log
- `app.log` - PHP-FPM log
- `cron.log` - Cron log
    - In Pro Staging and Production, this is present only in the first web node.
- `dns.log`
- `error.log` - NGINX error log
- `php.access.log` - PHP access log
- `php.debug.access.log` - PHP access debug log

Note that these correspond with the log types available to be viewed via the Cloud CLI tool.

Service Logs

The logs related to other services within the Cloud architecture - Redis, database, etc. - are not available in PaaS environments but are accessible in Pro Staging and Production.

- Redis log
    - Staging:
      `/var/log/platform/{project-id}_stg/redis-server-{project-id}_stg.log`
    - Production:
      `/var/log/platform/{project-id}/redis-server-{project-id}.log`

- Elasticsearch log
  - `/var/log/elasticsearch/elasticsearch.log`
- Java garbage collection log
  - `/var/log/elasticsearch/gc.log`
- Mail log
  - `/var/log/mail.log`
- MySQL error log
  - `/var/log/mysql/mysql-error.log`
- MySQL slow log
  - `/var/log/mysql/mysql-slow.log`
- RabbitMQ log
  - `/var/log/rabbitmq/rabbit@host1.log`

Pro Scaled Architecture

For Pro scaled architecture, logs corresponding with services on service nodes (e.g., MySQL) are found on those service nodes, not on the web node.

As a reminder, Pro plans include New Relic Logs, which provides a separate dashboard aggregating logs from all nodes.

> **Reminder**
>
> Remember that Pro plans include automatic log rotation and archiving. This is the case for all logs with a fixed filename, except deploy and post-deploy logs.

Further Reading

- Viewing and Managing Logs

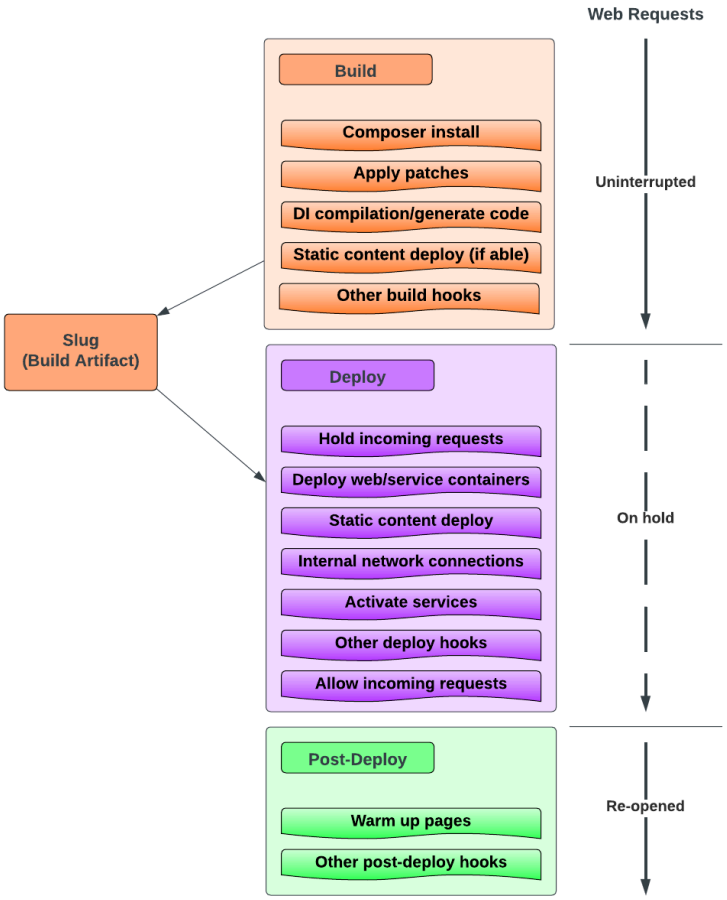
- Pro Scaled Architecture

7.06: Demonstrate the ability to deploy a project (Main steps of deployment)

In this section, we'll review the details of the steps that occur when the application is built and deployed in the Cloud environment.

Deployments consist of three phases:

- **Build**: The application is built. No services or connections are available to the build process, and incoming requests to the existing website continue.
- **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.



Build Phase

In the build phase, the Composer installation is run to assemble the full application files. In addition:

- Patches are applied.
- The `setup:di:compile` operation is run, creating all generated code files.
- If `app/etc/config.php` doesn't exist, it is generated, enabling all modules.
- If all necessary configuration is available on disk (more on this later), static content deployment is performed.

The final result of all these tasks is mounted on a read-only file system Cloud documentation refers to as a "slug".

The entire build phase takes place in a location isolated from the currently active application,

which continues to accept incoming web requests, undisturbed by the build. It's also important to understand that no services or network connections are available to the newly built slug during this phase; all operations must rely only on disk configuration.

Note that the build process can be run in a local development environment using the Cloud CLI tool command `local:build`. This command has a number of options:

- `--abslinks` / `-a` - Use absolute links
- `--source` / `-s` - Source directory. Defaults to current project root.
- `--destination` / `-d` - Destination directory, to which the web root of the app will be symlinked
- `--copy` / `-c` - Copy to build directory, instead of symlinking from source
- `--clone` - Use Git to clone current HEAD to build directory
- `--run-deploy-hooks` - Run deploy and/or post_deploy hooks
- `--no-clean` - Do not remove old builds
- `--no-archive` - Do not create or use a build archive
- `--no-backup` - Do not back up previous build
- `--no-cache` - Disable caching
- `--no-build-hooks` - Do not run post-build hooks
- `--no-deps` - Do not install build dependencies locally

Deploy Phase

In the deploy phase, containers and services are activated, the file system is mounted, and internal network connections between containers/nodes are opened. Other tasks that require access to services are then run.

- Updates to `env.php` resulting from deploy variables are done in this phase.
- If static content deployment couldn't be done during the build phase, it is performed now.

- `setup:upgrade` is run, making any schema or data changes in the database.

Because of the nature of the tasks performed in this phase, the site is put in maintenance mode, cron jobs and queue consumers are disabled, and external network requests are put on hold. This makes the deploy phase the determining factor in any site downtime occurring when a deployment is made, and the most critical factor in reducing such downtime is moving static content deployment to the build phase instead; we'll discuss this in more detail later.

Web requests coming in during the deploy phase are frozen for 60 seconds, so if static content deployment is avoided in this phase and there are no long-running operations resulting from `setup:upgrade`, it is possible to have zero perceptible downtime for the entire deployment.

Post-Deploy Phase

The post-deploy phase largely consists of whatever hooks you want to add to it. This is the phase for executing any tasks that should be performed after the application is fully built, all services are available, and network connections have resumed. Incoming traffic has been re-opened at this point, so it's important not to execute any tasks that would disrupt normal site operation.

There are few built-in Cloud tasks executed by default in the post-deploy phase, but these include re-enabling the cron and clearing the cache. Certain post-deploy variables will also trigger built-in tasks:

- Page cache is "warmed up" by issuing HTTP requests for pages defined in `WARM_UP_PAGES`.
- Details from any "time to first byte" tests required by `TTFB_TESTED_PAGES` are written to the log.

Hooks

While there are obviously critical tasks that are built into the Cloud infrastructure's execution of

the deployment phases, you also have the ability to define your own custom CLI commands to run.

The `hooks` property in the `.magento.app.yaml` file defines the commands to run during each phase.

```
hooks:
    build: |
        set -e
        composer install
        php ./vendor/bin/ece-tools run scenario/build/generate.xml
        php ./vendor/bin/ece-tools run scenario/build/transfer.xml
    deploy: |
        php ./vendor/bin/ece-tools run scenario/deploy.xml
    post_deploy: |
        php ./vendor/bin/ece-tools run scenario/post-deploy.xml
```

The above is the default configuration of `hooks`. You can see that the `composer install` command itself is included in the `build` phase. Many of the other core built-in tasks involved in each phase are explicitly run with the `ece-tools` scenarios seen in the configuration. The `set -e` command ensures that the phase fails immediately with the first failed command, rather than proceeding with the others.

You can add your own commands to this list of hooks, in the appropriate phase, and they will be run as part of the deployment.

> **Important**
>
> Just remember to consider the implications of each phase when determining the appropriate place for your custom commands!

- No services or connections are available during the build phase. Commands that need access to the database, for example, > will fail.
- The site is in maintenance mode during the deploy phase.
- All post-deploy hooks occur when the newly deployed application is "live."

Partial Deploys and Redeployment

The Cloud deployment process will not run the build phase if no changes affecting the application file state are detected. The "slug" from a pre-existing build will be reused, and only the deploy and post-deploy phases will be run. Updating environment variables with the Project Web Interface or Cloud CLI tool, for example, should trigger such a "partial" deploy.

You can perform a redeployment without any code or configuration changes using the Cloud CLI tool:

```
magento-cloud environment:redeploy
```

This also has the alias `redeploy`. Keep in mind, however, that this only performs a "partial" deployment consisting of the deploy and post-deploy phases. A new build is not performed.

In cases where a fresh build is desired but no code state has changed, we tend simply to use a comment in `magento-vars.php` with a numerical value that can be incremented for a "benign" new commit. However, you shouldn't make a routine of this kind of coerced re-build. Keep in mind that there's a reason new builds are not performed; the file state of the new build would be identical to the old!

Further Reading

- Deployment Overview

- Hooks Property
- Deployment Best Practices

Scenario-Based Deployments

Current versions of `magento/ece-tools` (2002.1.0 and later) support XML-based scenarios for performing various deployment tasks.

We've seen that the default `hooks` property in `.magento.app.yaml` makes use of these scenarios:

```
hooks:
    build: |
        ...
        php ./vendor/bin/ece-tools run scenario/build/generate.xml
        php ./vendor/bin/ece-tools run scenario/build/transfer.xml
    deploy: |
        php ./vendor/bin/ece-tools run scenario/deploy.xml
    post_deploy: |
        php ./vendor/bin/ece-tools run scenario/post-deploy.xml
```

These calls to `ece-tools run` pass an XML filename as an argument. The files seen above are part of the `ece-tools` package itself, located in the `scenario` directory. The scenarios defined in these XML files declare sequential steps and PHP classes to execute. The `post-deploy.xml` file, for example, contains the following:

```
<?xml version="1.0"?>
<scenario xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xsi:noNamespaceSchemaL
    <step name="is-deploy-failed" type="Magento\MagentoCloud\Step\PostDeploy\DeployFai
    <step name="validate-config" type="Magento\MagentoCloud\Step\ValidateConfiguration
        <arguments>
            <argument name="logger" xsi:type="object">Psr\Log\LoggerInterface</argumen
```

```
            <argument name="validators" xsi:type="array">
                <item name="warning" xsi:type="array">
                    <item name="debug-logging" xsi:type="object" priority="100">Magent
                </item>
            </argument>
        </arguments>
    </step>
    <step name="enable-cron" type="Magento\MagentoCloud\Step\PostDeploy\EnableCron" pr
    <step name="backup" type="Magento\MagentoCloud\Step\PostDeploy\Backup" priority="4
    <step name="clean-cache" type="CleanCache.PostDeploy" priority="500"/>
    <step name="warm-up" type="Magento\MagentoCloud\Step\PostDeploy\WarmUp" priority="
    <step name="time-to-first-byte" type="Magento\MagentoCloud\Step\PostDeploy\TimeToF
</scenario>
```

A deep dive into the core scenarios and their PHP classes is beyond the scope of this course, but it's important to understand how you can provide your own scenarios or override existing ones.

The `ece-tools run` command accepts multiple arguments, and if multiple filenames are provided, their XML is merged together for the final scenario configuration. For example:

```
hooks:
    post-deploy: |
        php ./vendor/bin/ece-tools run scenario/post-deploy.xml vendor/{vendor-name}/{
```

The core version of `post-deploy.xml` will be merged with the provided custom file. The last arguments have the highest priority.

As an example of what can be done with this extension method, consider the following:

```
<?xml version="1.0"?>
```

```
<scenario xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xsi:noNamespaceSchemaL
    <step name="validate-config" type="{VendorName}\{ModuleName}\Step\ValidateConfigur
    <step name="time-to-first-byte" type="Magento\MagentoCloud\Step\PostDeploy\TimeToF
</scenario>
```

The above XML in your own scenario file would entirely replace the PHP class to be executed as the `validate-config` step and would change the priority of the `time-to-first-byte` step to run before the `warm-up` step.

You can also add a `skip` attribute to an existing step to remove it:

```
<?xml version="1.0"?>
<scenario xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xsi:noNamespaceSchemaL
    <step name="backup" skip="true" />
</scenario>
```

You can also, of course, add separate `ece-tools run` commands to the configured `hooks` to run separate scenarios.

You can read more details about scenario-based deployments, including info on how to override the service contracts from the default `ece-tools` package, in the documentation.

Further Reading

- Scenario-Based Deployments

7.07: Define features provided by ECE tools

As we've seen, the package `magento/ece-tools` is included as a dependency of the main Cloud metapackage. `ece-tools` is a set of scripts and commands for managing various tasks within the Adobe Commerce application.

Cloud CLI vs ece-tools

With two different CLI tools involved as key components of Cloud management, the distinction between the two can be confusing. The Cloud CLI tool is a *standalone* tool, and while the local project can provide context for its operations, it is mainly designed for managing high-level tasks in the Cloud environments remotely. `ece-tools` is a dependency installed as *part of* the Adobe Commerce application and provides lower level functions designed for running in the context of the application itself.

Basics

As a package installed within the Adobe Commerce application, the `ece-tools` command can be run directly from the `vendor` directory in any active environment:

```
php vendor/bin/ece-tools
```

A list of all available commands can be viewed with:

```
php vendor/bin/ece-tools list
```

Build and Deploy

Many of the utilities in `ece-tools` directly control the building and deployment of the application in the Cloud environment. We'll become more familiar with the build and deploy process later. You will not typically interact directly with these commands, as they are used automatically during automated Cloud deployments, based on configured hooks.

Command	Description
build	Builds application.
deploy	Deploys the application.
patch	Applies custom patches.
post-deploy	Performs after deploy operations.
build:generate	Generates all necessary files for build stage.
build:transfer	Transfers generated files into init directory.

With the exception of `patch`, the commands seen above are shortcuts for the more generic `run` command, which executes XML-based scenarios defining complex tasks. For example:

```
php vendor/bin/ece-tools run scenario/deploy.xml
```

The `error:show` command can be used to view info about errors from the most recent deployment.

Application/Environment Configuration

The following `ece-tools` commands deal with managing various pieces of configuration for the application or environment.

These commands deal specifically with configuration written to the `.magento.env.yaml` file:

Command	Description	Notable Arguments/ Options
cloud:config:create	Creates a `.magento.env.yaml` file with the	`<configuration>`

Command	Description	Notable Arguments/ Options
	specified build, deploy, and post-deploy variable configuration.	- JSON config
cloud:config:update	Updates the existing `.magento.env.yaml` file with the specified configuration.	`<configuration>` - JSON config
cloud:config:validate	Validates `.magento.env.yaml` configuration file	

`ece-tools` includes its own command for dumping current Store Configuration settings to disk (in `app/etc/config.php` and `app/etc/env.php` files):

Command	Description
config:dump	Dump configuration for static content deployment.

And finally, there is a command for viewing various configuration in the current environment:

Command	Description
env:config:show	Display encoded cloud configuration environment variables.

`env:config:show` accepts a `variable` argument with one of three values: "services", "routes", or "variables." This results in the output of settings related only to, respectively, the services configured in the current environment, the configured routes, and environment variables values that are applied.

Managing Cron

The following commands allow management of the Magento cron configuration in the environment.

Command	Description	Notable Arguments/Options
cron:disable	Disable all Magento cron processes and terminates all running processes.	
cron:enable	Enables Magento cron processes.	
cron:kill	Terminates all Magento cron processes.	
cron:unlock	Unlock cron jobs that stuck in "running" state.	`--job-code` - Cron job to unlock

Smart Wizards

Several `ece-tools` commands are used to verify that the state of certain environment variables are optimized for certain scenarios. Re-review this list once you understand more about the environment variables mentioned!

Command	Description
wizard:ideal-state	Check that `SCD` is on the build stage, the `SKIP_HTML_MINIFICATION` variable is `true`, and the post_deploy hook configured in the Cloud environment.
wizard:master-slave	Check that the `REDIS_USE_SLAVE_CONNECTION` variable and the `MYSQL_USE_SLAVE_CONNECTION` variable is `true`.
wizard:scd-on-demand	Check that the `SCD_ON_DEMAND` global environment variable is `true`.

Command	Description
wizard:scd-on-build	Check that the `SCD_ON_DEMAND` global environment variable is `false` and the `SKIP_SCD` environment variable is `false` for the build stage. Verifies that the `config.php` file contains information for stores, store groups, and websites.
wizard:scd-on-deploy	Check that the `SCD_ON_DEMAND` global environment variable is `false` and the `SKIP_SCD` environment variable is `false` for the deploy stage. Verifies that the `config.php` file does NOT contain the list of stores, store groups, and websites with related information.

Updating `ece-tools`

It's important to keep the `ece-tools` package up to date in your project, for maximum deployment efficiency and for up-to-date patches.

Updating is intuitive, since the package is updated just like any other Composer dependency:

```
composer update magento/ece-tools --with-dependencies
```

Further Reading

- ECE-Tools
- Updating ECE-Tools
- Smart Wizards

7.08: Identify uses for ECE patches (Security breach)

Patches

The Cloud build process automatically applies code patches from various sources after the initial Composer installation. These include critical security patches and optional quality patches provided by the core packages, as well as support for applying your own custom patches.

The Packages

The `magento/magento-cloud-patches` package, which is a dependency of `magento/ece-tools`, contains the CLI command used to apply patches, as well as containing required security patches themselves.

The `ece-tools` package also depends on `magento/quality-patches`, which contains available *optional* quality patches.

Required security patches are automatically applied when the patch operation is run during the build phase of deployments.

Because of the critical nature of required security patches, it is highly recommended to upgrade the `ece-tools` package and its dependencies on a regular basis, to ensure the latest security patches have been applied in your project.

Reminder

The `ece-tools` package is updated like any other Composer package:

```
composer update magento/ece-tools --with-dependencies
```

Optional Quality Patches

Use the build variable `QUALITY_PATCHES` in `.magento.env.yaml` to specify any official quality patches you want applied in your project.

```yaml
stage:
  build:
    QUALITY_PATCHES:
      - MC-31387
      - MDVA-4567
      - MC-456345
```

Any patches you declare here will also be automatically applied by the patch operation.

Available quality patches can be viewed in the Cloud documentation.

Custom Patches

The `m2-hotfixes` directory in the project root is reserved for any additional patch files that should be applied to your application files.

Simply add patch files to this directory, and they will also be automatically applied by the patch operation.

> **Core Concept**
>
> Make sure file paths in `m2-hotfixes` patches are relative to the project root. Patches for files in Composer packages, for example, should include the full file path including `vendor/{vendor-name}/{module-name}`.

Patch Order

Patches are applied in the following order:

1. Required patches
2. Selected quality patches
3. Custom patches from `m2-hotfixes`

The `ece-patches` Command

The `ece-patches` CLI command (provided by `magento/magento-cloud-patches`) is the command used to apply patches automatically as part of the Cloud deployment process. It can also be used manually. In any working environment, you can run the following to see all available patches:

```
php vendor/bin/ece-patches status
```

You can also use the command to manually apply patches in a local development environment, exactly as they are applied in Cloud builds:

```
php vendor/bin/ece-patches apply
```

If you have a reason to un-do patches in a local development environment, you can do so with `revert`:

```
php vendor/bin/ece-patches revert
```

Further Reading

- Applying Patches

7.09: Describe how to Maintain and upgrade ECE tools

Upgrading ECE tools is very easy:

```
composer update magento/ece-tools --with-dependencies
```

7.10: Distinguish when to contact support yaml files and limitations (DIY vs Support tickets)

Everything discussed thus far is changeable in all environments, with a couple of major exceptions in Pro stating and production environments.

Pro environments (staging and production only)

Adobe recommends making changes to the below files and deploying to an Integration environment to make sure it works as expected. Then, deploy to Staging and make sure it doesn't work.

The helpful part of this procedure is your configuration is tracked in version control. The unhelpful part is you can't verify that the change was properly made.

These are the changes that must be made manually by Adobe support:

- Adding or modifying services in `.magento/services.yaml`
- `disk`, `mounts`, `cron`, in `.magento.app.yaml`
- `routes.yaml` (routes and redirects)

7.11: Demonstrate basic knowledge of OOTB FASTLY features configuration and installation

Fastly is a large topic. You're encouraged to read the full available documentation, but here we'll cover the key concepts that are likely to show up on your Adobe Commerce certification exams.

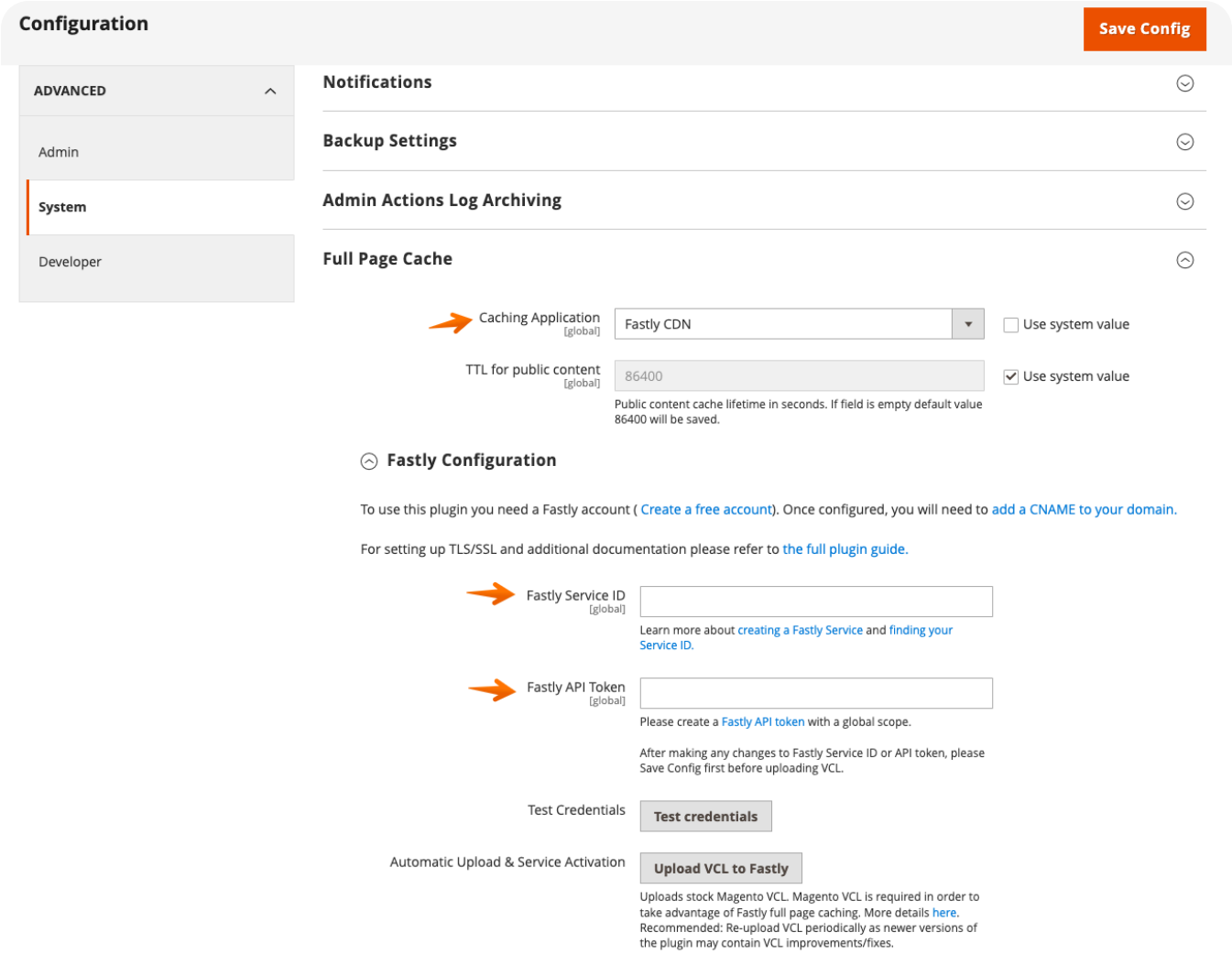
Fastly sits in front of your web application as a content delivery network (CDN) and proxies all incoming requests. (Your site DNS records will actually point traffic to Fastly servers rather than directly to the servers hosting the Adobe Commerce application.) Fastly is fully optimized for fast serving of static assets and processing/resizing of images, and it also caches full page responses, allowing many web requests to be satisfied without ever hitting the Magento application at all.

The Fastly CDN module for Magento is automatically installed in the initial codebase of a Cloud project and makes configuration and management of Fastly through the Magento admin panel fairly seamless.

Credentials and Enabling

Fastly should be automatically configured on Pro Staging and Production environments, and Starter Production environments, as part of the initial Cloud on-boarding process. Starter also provides a set of Fastly credentials for Staging; since the Staging environment is not automatically provisioned, you can obtain this separate set of credentials and configure Fastly once you have created it.

To integrate with Fastly, Adobe Commerce needs a Fastly API token and service ID. In Stores > Configuration > Advanced > System > Full Page Cache, the "Caching Application" setting must be set to "Fastly CDN", and the token and service ID must be entered in the appropriate fields in the Fastly Configuration subsection.



Better than configuring Fastly in the admin, however, is to lock these configs, which can be done with the following environment variables. (We'll discuss the use of environment variables in a later section.)

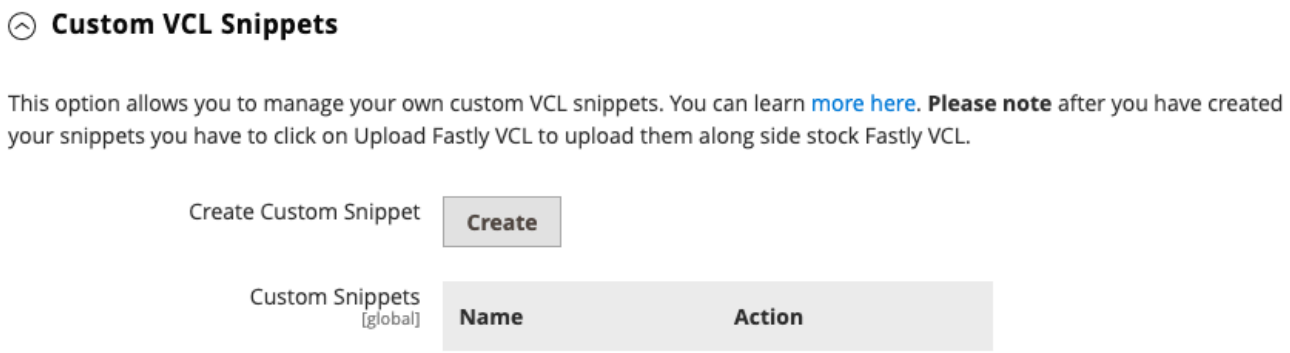
- `CONFIG__DEFAULT__SYSTEM__FULL_PAGE_CACHE__CACHING_APPLICATION` (value 42)
- `CONFIG__DEFAULT__SYSTEM__FULL_PAGE_CACHE__FASTLY__FASTLY_API_KEY`
- `CONFIG__DEFAULT__SYSTEM__FULL_PAGE_CACHE__FASTLY__FASTLY_SERVICE_ID`

In addition to preventing accidental changes that will break your application, locking caching/Fastly configs has the added benefit of allowing Staging credentials to remain configured even if Production data is synced down.

You should find that these settings are configured when your Cloud environment is provisioned.

VCL Configuration

Fastly requires appropriate VCL configuration, specific to Magento, in order to properly hash, cache and pass through web requests. The same Store Configuration section referenced above contains an "Upload VCL to Fastly" button that will automatically upload the stock VCL files from the Magento codebase. (And a Custom VCL Snippets section allows you to augment this with your own configuration.)



SSL and DNS

In order for secure HTTPS traffic to be handled by Fastly, domain-validated SSL/TLS certificates are required. During the Cloud on-boarding process, a Let's Encrypt certificate will be provided for the site's main domain. (For Pro, a separate certificate will also be provided for Staging.)

The on-boarding process will include instructions for achieving domain validation with CNAME records in your DNS configuration.

In order to facilitate pre-launch development and setup, SSL and Fastly configuration is set up to support the domain `mcprod.{your-domain}.com` (as well as `mcstaging.{your-domain}.com` for Pro). You must configure CNAME records in your DNS settings to point to the appropriate Fastly endpoint for the above subdomains, as well as for your main domain at

launch.

Domain	CNAME
mcprod.your-domain.com	prod.magentocloud.map.fastly.net
mcstaging.your-domain.com	prod.magentocloud.map.fastly.net

Security Features

Fastly provides a web application firewall (WAF) for the Cloud Production environment, which provides PCI-compliant protection from malicious traffic.

Also included are built-in DDoS protection and origin cloaking. Read more about these features in the documentation.

Advanced Configuration

There are a host of other Fastly configuration options that can be managed directly in the Magento admin, from Stores > Configuration > Advanced > System > Full Page Cache > Fastly Configiuration. These include settings like purge options, GeoIP handling, timeout settings, and more.

Review the collection of advanced settings here:

Advanced Configuration

Force TLS

Enable/Disable **Current state:** unknown

Return a **301 Moved Permanently** redirect to any unencrypted request, and redirect to the TLS equivalent. Also sets HSTS headers.

Admin path timeout (1-600) [global]

180

Use system value

Time in seconds for the admin path first byte timeout. Please reupload VCL after making changes.

Ignored URL Parameters [global]

utm_.*, gclid, gdftrk, _ga, mc_.*, trk_.*, dm_i, _ke, sc_

Use system value

A comma separated list of ignored query string parameters. Please reupload VCL after making changes.

Stale Content Delivery Time [global]

86400

Use system value

Time in seconds that Fastly will serve stale content while fresh content is being requested. Set to 0 to disable this feature.

Stale Content Delivery Time in Case of Backend Error [global]

86400

Use system value

Time in seconds that Fastly will continue to serve stale content if your origin is unavailable. Set to 0 to disable this feature.

X-Magento-Tags Size [global]

16383

Use system value

Maximum X-Magento-Tags header size (in bytes).

Purge Category [global]

No

Choose to purge all the category assets when saving a change to that category.

Purge Product [global]

Yes

Choose to purge all the product's assets when saving a change to that product.

Purge CMS Page [global]

No

Choose to purge page content when updating or adding a new page in the Magento CMS.

Preserve Static Assets on Purge [global]

Yes

When flushing cache, flush only dynamic content and preserve static assets.

Use Soft Purge [global]

Yes

Soft Purge needs to be turned on in order to serve stale content.

Enable GeoIP [global]

No

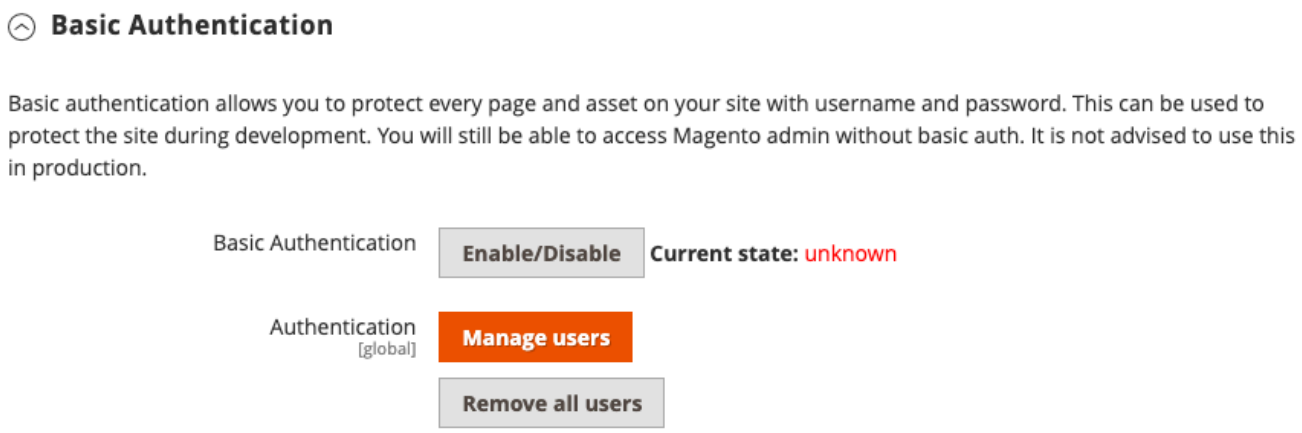
Enable GeoIP for country/language lookup.

Enable Fastly Edge Modules [store view]

Yes

Enables/Disables Fastly Edge Modules menu.

Other available settings include Basic Authentication:



Blocking:

Blocking

Blocking

Enable/Disable **Current state:** unknown

Block traffic from selected ACL IPs and Countries. Please Save Config prior to enabling.

Update Blocking

Update Blocking Config

Applies blocking changes to active setup. HAS TO BE clicked anytime you make blocking changes.

Blocking Type
[global]

Blocklist

Blocklist: block ALL access for users from selected countries/ACLs
Allowlist: block ALL access EXCEPT for users from selected countries/ACLs

Country List
[global]

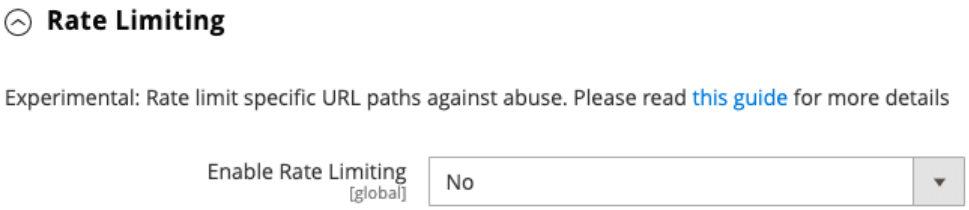
Afghanistan
Åland Islands
Albania
Algeria
American Samoa
Andorra
Angola
Anguilla
Antarctica
Antigua & Barbuda

List of countries that can be selected. Multiple selections allowed.

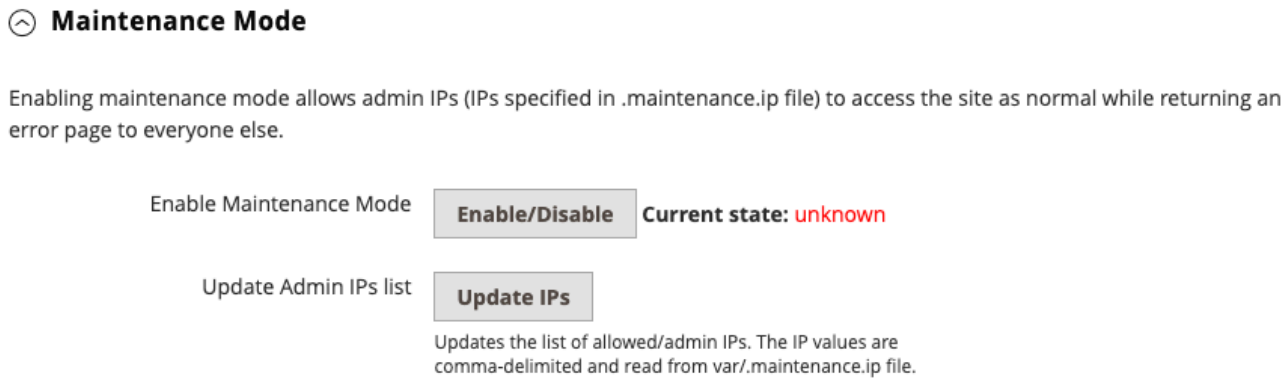
ACLs
[global]

List of ACLs that can be selected. Multiple selections allowed.

Rate Limiting:



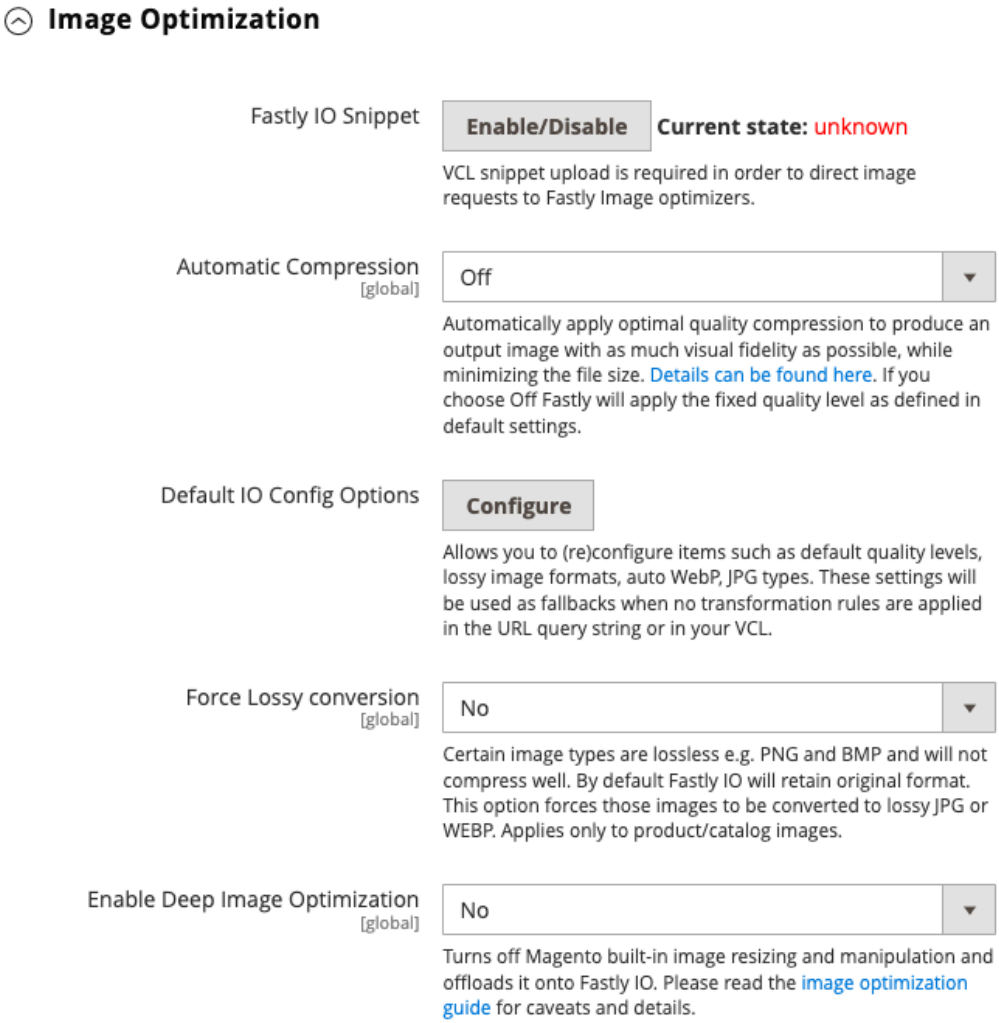
And Maintenance Mode:



The admin Cache Management page also features actions for directly purging Fastly of specific content types.

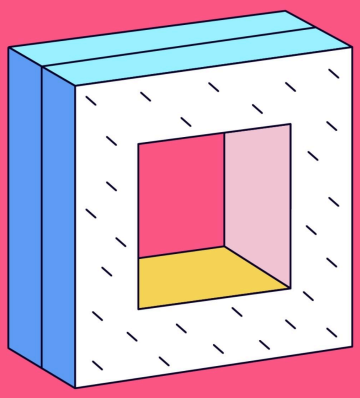
Image Optimization

In the Magento admin, in Stores > Configuration > Advanced > System > Full Page Cache > Fastly Configuration, there is a section for Image Optimization. From here, Fastly IO can be enabled (which results in uploading the appropriate VCL snippet), and the dynamic resizing of images on your site will be offloaded to Fastly.



Further Reading

- Adobe Commerce Fastly Documentation



Objective 8

Setup/Configuring Adobe Commerce Cloud

10% of the test / 7 questions

Adobe Commerce Expert Developer Prep Guide, AD0-E716

SwiftOtter

8.01: Describe how to setup/configure Adobe Commerce Cloud

For our discussions in this chapter, it's important to be clear on the distinction between the *application* and *services*.

We've talked about the various services that are or can be installed and configured in your application's environment containers - services like MySQL, Redis, etc. These services are obviously critical to the Magento application but are not considered *part* of it from the standpoint of the Cloud infrastructure's configuration.

By contrast, NGINX, PHP-FPM, and system details like crons and disk mounts comprise the application itself. The application has access to and uses services.

Let's take a tour of the main configuration file that controls the application definition.

Main Config File

The primary configuration of the Cloud application is handled in the file `.magento.app.yaml`. This is automatically created and configured with defaults when the project is provisioned.

Basic Application Details

The application is given a name ("mymagento" by default), which is used in other configuration (namely, config for routes). It's important that the application name not be changed after initial deployment.

```
name: mymagento
```

The application is a PHP application, and the `type` property is used to declare this and the PHP version. The `build` and `flavor` properties declare core build tasks, but `flavor` should be set to "none" for the default Composer 2 installation to be used.

```yaml
type: php:8.1
build:
    flavor: none
```

The PHP version deployed into the Cloud environments can be changed with `type`.

> **Note**
>
> The versions of NGINX and the operating system cannot be changed via configuration.

The `dependencies` property controls PHP, Ruby or Node.js dependencies the application needs, which become available in the `PATH` during the build process and runtime. This property will automatically be configured to include Composer as a dependency.

```yaml
dependencies:
    php:
        composer/composer: '2.2.4'
```

PHP extensions can be enabled and disabled with the `runtime` property.

```yaml
runtime:
    extensions:
        - xsl
        - newrelic
        - sodium
    disabled_extensions:
        - soap
```

The documentation contains a list of PHP extensions that are supported, enabled by default, and not allowed to be disabled.

> **Important**
>
> Enabling the `ioncube` or `sourceguardian` extensions in Pro requires submitting a support ticket.

Cron Configuration

Cron jobs to run in the Cloud environment can be configured in the YAML file. There will be configuration already set up for the default Magento cron job.

```
crons:
    cronrun:
        spec: "* * * * *"
        cmd: "php bin/magento cron:run"
```

Other Configuration

We will delve into the use of the `relationships`, `web`, `disk`, `mounts` and `hooks` properties in later lessons.

The `.magento.app.yaml` file is one of several locations where environment variables can be defined, using the `variables` property. This is appropriate for variables whose values should be considered fixed application configuration applicable to all environments, and a couple of specific variables will be configured already.

```
variables:
```

```
    env:
        CONFIG__DEFAULT__PAYPAL_ONBOARDING__MIDDLEMAN_DOMAIN: 'payment-broker.magento.
        CONFIG__STORES__DEFAULT__PAYPAL__NOTATION_CODE: 'Magento_Enterprise_Cloud'
```

As we have seen, all users with the Contributor or role or higher for an environment are able to SSH into that environment. Application configuration can be used to change that minimum required role:

```
access:
    ssh: viewer
```

For Starter plans only, the main configuration file can be used to configure rules for the outbound firewall. You can find full information on this configuration in the documentation.

Another property not initially included in the `.magento.app.yaml` file is `workers`, used to configure worker instances independent of the web process. See the documentation for full details.

Further Reading

- Properties Summary
- Variables
- PHP Settings
- Firewall
- Workers

Disk Configuration

The `.magento.app.yaml` config file supports options for managing disk allocation and mounts for PaaS environments.

> **Important**
>
> The configuration we're discussing in this section - the use of both the `disk` and `mounts` properties - does not automatically take effect in Pro Staging and Production environments. A support ticket must be opened in order to update disk configuration in these environments to match `.magento.app.yaml`.

Disk Allocation

The total disk size of the application can be set with the `disk` property, with a value expressed in MB.

```
disk: 20480
```

The minimum recommended disk size is 256MB.

Mounts Configuration

File mounts in the Cloud environments are shared volumes that are writable (unlike the rest of the application directory) and whose files will persist across deployments. By default, the following directories are configured as mounts in `.magento.app.yaml`:

- `var`
- `app/etc`
- `pub/media`
- `pub/static`

You can use the `mounts` property to configure additional directories as mounts.

```
mounts:
    "var": "shared:files/var"
    "app/etc": "shared:files/etc"
    "pub/media": "shared:files/media"
    "pub/static": "shared:files/static"
    "custom": "shared:files/custom"
```

> **Note**
>
> The Cloud documentation mentions the usage of a `disk` property to define the allocated size of a particular mount, but it's unclear how this fits in with the `mounts` syntax!

Mount Details with Cloud CLI

Several commands for viewing and managing mounts are available with the Cloud CLI tool.

Command	Description	Notable Arguments/Options
mount:download	Download files from a mount, using rsync	`--all` / `-a` - Download from all mounts `--mount` / `-m` - Mount to download `--target` - Local directory to sync to `--source-path` - (When using `--all`) Uses source path instead of mount path as subdirectory (e.g., `media` instead of `pub/media`) `--delete` - Delete local files that don't exist remotely `--exclude` - Files to exclude `--include` - Files to include

Command	Description	Notable Arguments/Options
mount:list (mounts)	Get a list of mounts	
mount:size	Check the disk usage of mounts	
mount:upload	Upload files to a mount, using rsync	`--source` - Local directory to copy `--mount` / `-m` - Mount to sync to `--delete` - Delete remote files that don't exist locally `--exclude` - Files to exclude `--include` - Files to include

`mount:list` will show you the details of the existing mounts.

```
chrisnanninga@Chriss-MBP:02:39 PM:[redacted] $ magento-cloud mount:list
Mounts on [redacted]-integration1-[redacted]--mymagento@ssh.us-4.magento.cloud:
+------------+--------------------+
| Mount path | Definition         |
+------------+--------------------+
| var        | source: local      |
|            | source_path: var   |
| app/etc    | source: local      |
|            | source_path: etc   |
| pub/media  | source: local      |
|            | source_path: media |
| pub/static | source: local      |
|            | source_path: static |
+------------+--------------------+
```

And `mount:size` will display the details of allocated and available disk size for mounts.

```
chrisnanninga@Chriss-MBP:02:43 PM:                                    $ magento-cloud mount:size
Checking disk usage for all mounts on                -integration1-          --mymagento@ssh.us-4.magento.cloud...

+------------+-----------+----------+-----------+-----------+--------+
| Mount(s)   | Size(s)   | Disk     | Used      | Available | % Used |
+------------+-----------+----------+-----------+-----------+--------+
| app/etc    | 216 KiB   | 19.6 GiB | 744.0 MiB | 18.9 GiB  | 3.7%   |
| pub/media  | 161.8 MiB |          |           |           |        |
| pub/static | 4 KiB     |          |           |           |        |
| var        | 554.6 MiB |          |           |           |        |
+------------+-----------+----------+-----------+-----------+--------+
```

The `mount:download` and `mount:upload` commands can be used to `rsync` files from the remote mount to a local location or vice versa. If appropriate options (see above) are not provided, interactive prompts will allow you to specify the remote mount and local directory.

Further Reading

- `.magento.app.yaml` Properties
- Managing Disk Space

8.02: Apply Basic Cloud troubleshooting knowledge (Hierarchy of web UI and variables, configurations precedence)

The subject of variables in Cloud and the role they play throughout the infrastructure is multi-faceted and can quickly get confusing. There are different classifications of variables for different purposes, as well as different ways to set them. Some variables can be set in multiple ways (in which case there is an override priority), and others cannot. Some variables are available persistently as environment variables on the web server (e.g., can be `echo`'d in an SSH session), and some are not!

These are the major classifications of variables:

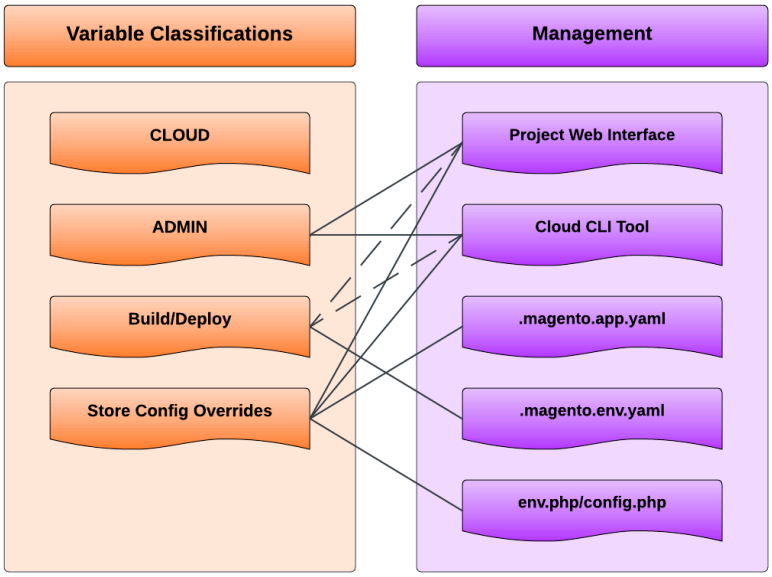
- CLOUD variables with various information about the Cloud project and infrastructure (read-only)
- ADMIN variables defining a main admin user and admin URL

- Build/deploy variables controlling various aspects of the deployment process
- Environment variables that override Magento Store Config settings

And variables can be set in these ways:

- The Project Web Interface (ADMIN and Store Config overrides)
- The Cloud CLI Tool (ADMIN and Store Config overrides)
- The `.magento.app.yaml` file (Store Config overrides)
- The `.magento.env.yaml` file (Build/deploy variables)

With the override of Magento Store Config settings in play, this means that the `app/etc/config.php` and `env.php` files, and values in the Magento database, are also relevant to priority order (though not specifically part of the topic of Cloud infrastructure).



In the diagram above, the dashed lines represent a management method that only works in *some* cases.

We have already seen a few examples of variables used in different contexts:

- `CONFIG__DEFAULT__SYSTEM__FULL_PAGE_CACHE__CACHING_APPLICATION`

    - This is an example of overriding a Magento Store Config setting, in this case setting the full page cache method.
- `COMPOSER_AUTH`
    - Another example of an environment variable, but a one-off example that doesn't correspond with a Store Config setting.
- `php:newrelic.license`
    - Also set as an environment variable, also not corresponding with a Store Config setting, but not directly available on the server.
- `MAGENTO_CLOUD_RELATIONSHIPS` and `MAGENTO_CLOUD_ROUTES`
    - Read-only settings with information about configuration.

8.03: Demonstrate understanding of cloud user management and onboarding UI

All aspects of managing your Cloud project require varying levels of user authorization. Let's take some time to become familiar with managing users, as well as the various roles and permissions required for different tasks.

The **Account Owner** role is assigned to only one user: the person who initially registered the account for the Cloud project. This role can perform any operation in the project and in any environment. It requires a support ticket to change the Account Owner.

All other users will have one of these two roles at the *project* level:

- **Viewer** is the de facto role. This role does not have access to project-level settings, and the user's permissions for specific environments will depend on their environment-level roles.
- **Super User** (called "admin" in the Cloud CLI tool) gives a user access to project-level setting and the management of other users (including other Super Users). Users with this role do not require environment-level roles and can perform any task in any environment.

If a user has the Viewer role at the project level, they can then be given a specific role for any given environment:

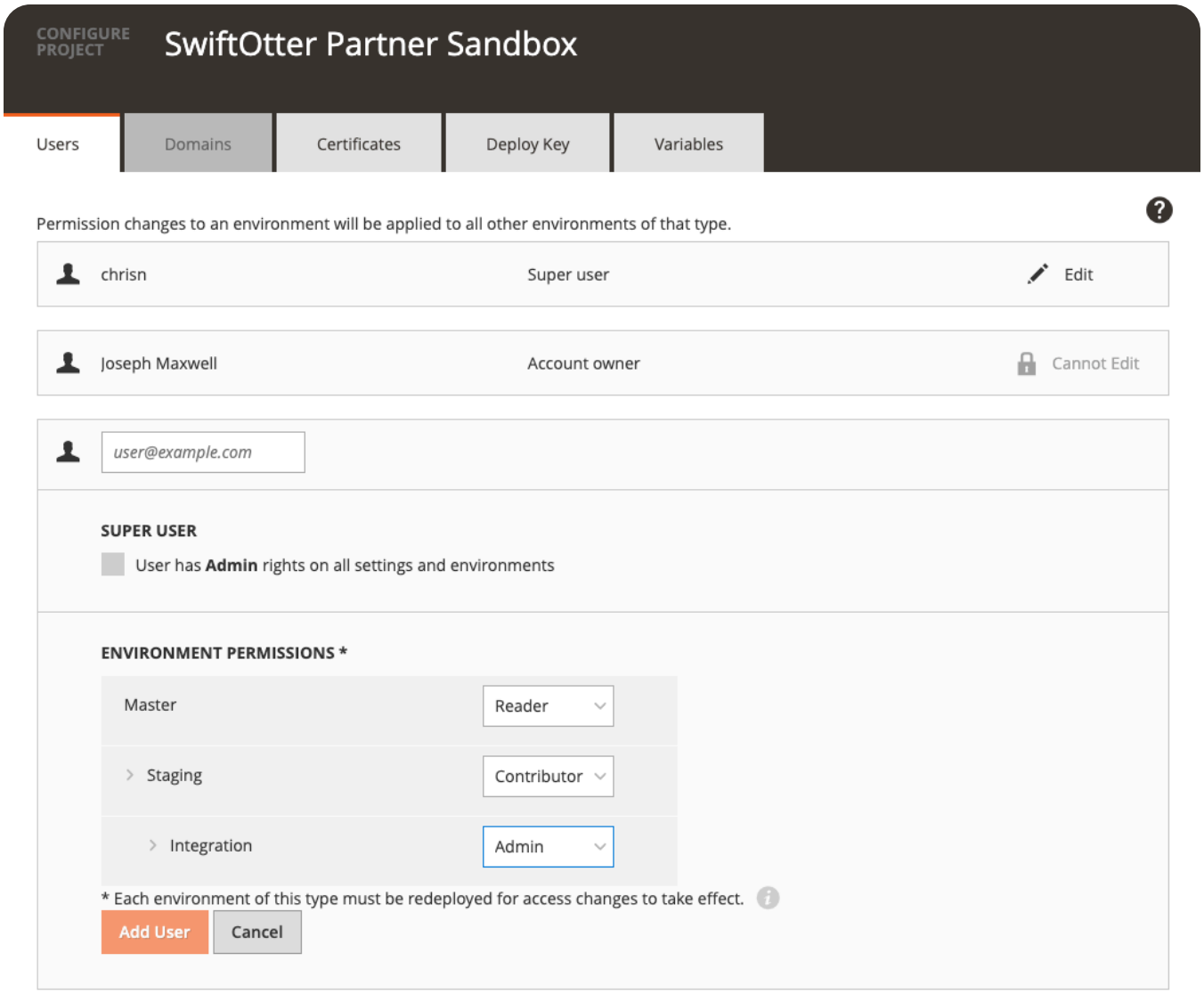
- **None**: A user can be denied any access to an environment at all. They will see the environment in the hierarchy in the Project Web Interface but will not be able to navigate to it or see any other details.
- **Viewer**: (Called "Reader" in the Project Web Interface.) The user can view the environment, including messages and logs in the Project Web Interface, but cannot perform any other tasks.
- **Contributor**: The user has SSH access to the environment, can push code changes to the branch, and can branch other environments from it.
- **Admin**: In addition to the permissions of a Contributor, the user can create snapshots of the environment and can manage settings. If the user also has Viewer access on the parent environment, they can also sync from that environment. And if they have Contributor access to the parent environment, they can merge to it.

> **Reminder**
>
> Remember that, for SSH access, your user account must have a public key associated. You can add a key in the web account management interface, or via the `ssh-key:add` command with the Cloud CLI tool.

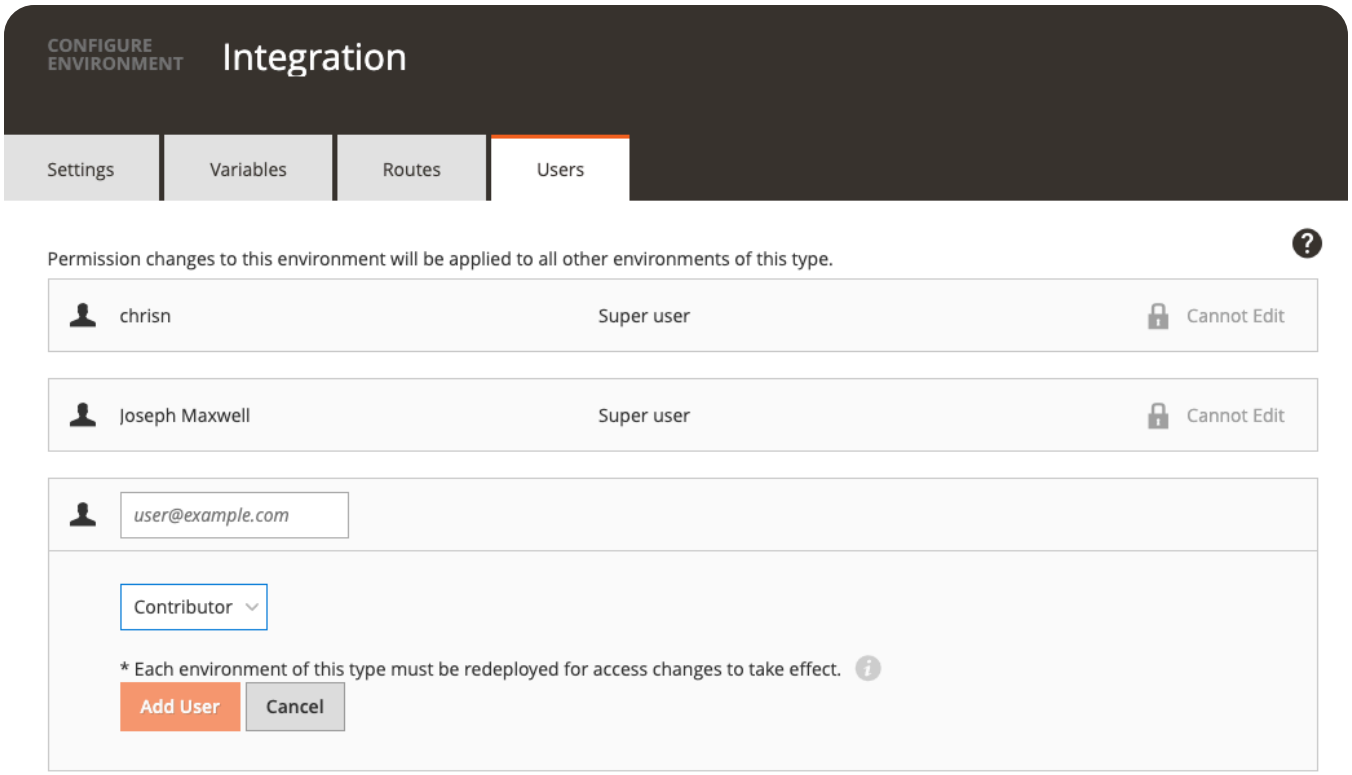
Managing Users in the Project Web Interface

As an Account Owner or Super User, you can manage other users in the Project Web Interface.

By using the "gear" icon to access project settings, you can edit users at the project level. Here, you can give users the Super User role, or set their roles for every specific environment.



You can also manage user roles for a specific environment with the Users tab in "Configure environment".



Users added in the environment interface will automatically receive the Viewer role at the project level.

Managing Users with the Cloud CLI Tool

A user can be added via the Cloud CLI tool:

```
magento-cloud user:add -r staging:contributor <email>
```

The above example creates a user (implicitly with the Viewer role at the project level) with the Contributor role on the Staging environment and None on all others. The roles that can be assigned after the ":" include "none", "viewer", "contributor" and "admin".

The option value before the ":" refers not to a specific environment but to a "type" of environment. The type "production" will apply to the Production environment, "staging" will apply to "Staging" (even on Starter, where it's up to you to create an environment with this name), and "development" will apply to all others.

A user can also be created with a project-level role:

```
magento-cloud user:add -r admin
```

The above example creates a Super User. "viewer" can also be specified.

If you run `user:add` without any parameters, a series of interactive prompts will allow you to specify the email address, the project-level role and (if not a Super User) the roles for each environment type.

```
chrisnanninga@Chriss-MBP:12:02 PM:                                    $ magento-cloud user:add
Enter the user's email address: user@example.com

The user's project role can be admin (a) or viewer (v).

Project role (default: viewer) [a/v]: viewer

The user's environment type role(s) can be admin (a), viewer (v), contributor (c) or none (n).

Role on type development (default: none) [a/v/c/n]: admin
Role on type production (default: none) [a/v/c/n]: viewer
Role on type staging (default: none) [a/v/c/n]: contributor

Adding the user user@example.com to SwiftOtter Partner Sandbox (              ):
  Project role: viewer
    Role on type development: admin
    Role on type production: viewer
    Role on type staging: contributor
```

Other commands exist for viewing and updating users.

Command	Description	Notable Arguments/Options
user:delete	Delete a user from the project	`<email>` - Email address
user:get	View a user's role(s)	`<email>` - Email address `--level` / `-l` - "project" or "environment"
user:list (users)	List project users	

Command	Description	Notable Arguments/Options
user:update	Update user role(s) on a project	`<email>` - Email address `--role` / `-r` - Role (same format as for `user:add`

Further Reading

- Manage User Access

8.04: Describe how to update cloud variables using UI

We'll refer to those variables that can be set with Project Web Interface or the Cloud CLI tool as "environment variables", even though they are not in all cases available as environment variables in the web server.

Variable names prefixed with `env:` are available as environment variables on the server, and this is how Magento Store Config overrides are set. This doesn't apply to certain variables, however, including the ADMIN vars.

Priority

Variables can be set at the project level or at the environment level. And environments, of course, have a hierarchical relationship.

When adding a variable to an environment, you can choose whether it can be inherited. For any given environment, a value set at that environment's level takes highest priority. If this doesn't exist, any inheritable values from parent environments are used. And if these don't exist either, any project-level value is used.

Properties

In addition to a name and a value, a variable can be given other properties:

- **Enabled**: The variable can be disabled (and thus ignored) without actually deleting it.
- **Inheritable**
- **Sensitive**: Indicates the variable contains a sensitive value.
- **JSON Value**: Indicates the variable's value is in JSON format.
- **Visible During Build**: Can be accessed as an environment variable on the server during the application build.
- **Visible at Runtime**: Can be accessed as an environment variable on the server at runtime.

> **Note**
>
> There is not extensive documentation on all of these properties, and sometimes their use is opaque. It's unclear, for example, what behavior would be affected by "JSON Value". You might expect the "Sensitive" property to result in a value being encrypted with the environment encryption key, but it does not. This property only results in obfuscating the value in the Project Web Interface and when viewed with the Cloud CLI tool.

Using the Project Web Interface

In the Project Web Interface, a Variables tab is displayed both when viewing project-level settings and when navigating to "Configure environment" for a specific environment.

In either location, you can create, view and edit variables.

Some of the variable properties can be set only at the project level, and some only at the environment level. It's not always clear why. (The ability to declare a variable "Sensitive", for example, should be a legitimate option at either the project or environment level, but this can only be set on an environment.)

- Project level: JSON Value, Visible During Build, Visible at Runtime

- Environment level: JSON Value, Enable, Inheritable, Sensitive

Using the Cloud CLI Tool

The Cloud CLI tool has a set of commands for managing variables.

Command	Description	Notable Arguments/Options
variable:create	Create a variable	`--level` / `-l` - "project" or "environment" `--name` - Variable name `--value` - Variable value `--json` - JSON-encoded value `--sensitive` - Sensitive value `--prefix` -- "none" or "env" `--enabled` - Whether var is enabled `--inheritable` - Inheritable by child environments `--visible-build` - Visible during build `--visible-runtime` - Visible at runtime
variable:delete	Delete a variable	`--level` / `-l` - "project" or "environment"
variable:get (vget)	View a variable	`--property` / `-P` - View a single property `--level` / `-l` - "project" or "environment"
variable:list (variables, var)	List variables	`--level` / `-l` - "project" or "environment"
variable:update	Update a variable	`--level` / `-l` - "project" or "environment" `--value` - Variable value `--json` - JSON-encoded value

Command	Description	Notable Arguments/Options
		`--sensitive` - Sensitive value `--enabled` - Whether var is enabled `--inheritable` - Inheritable by child environments `--visible-build` - Visible during build `--visible-runtime` - Visible at runtime

`variable:list` will display names, levels and values of all variables relevant to the specified (or inferred) environment:

```
chrisnanninga@Chriss-MBP:01:43 PM:~/SwiftOtter/Code/so-cloud-sandbox$ magento-cloud variable:list -e Integration1
Variables on the project SwiftOtter Partner Sandbox (3zltyuobw3r2q), environment Integration1:
+-----------------------------------------------+-------------+-------------------------------------------------------------------------------------------------------------+---------+
| Name                                          | Level       | Value                                                                                                       | Enabled |
+-----------------------------------------------+-------------+-------------------------------------------------------------------------------------------------------------+---------+
| ADMIN_EMAIL                                   | project     |                                                                                                             |         |
| ADMIN_FIRSTNAME                               | project     |                                                                                                             |         |
| ADMIN_LASTNAME                                | project     |                                                                                                             |         |
| ADMIN_USERNAME                                | project     |                                                                                                             |         |
| env:COMPOSER_AUTH                             | project     |                                                                                                             |         |
| ADMIN_URL                                     | environment | backend                                                                                                     | true    |
| env:CONFIG__DEFAULT__DEV__STATIC__SIGN        | environment | 1                                                                                                           | true    |
| env:CONFIG__DEFAULT__GENERAL__STORE_INFORMATION__NAME | environment | SwiftOtter Store                                                                                            | true    |
| env:CONFIG__DEFAULT__TWOFACTORAUTH__GENERAL__ENABLE | environment | 0                                                                                                           | true    |
| env:CONFIG__DEFAULT__WEB__SECURE__USE_IN_ADMINHTML | environment | 1                                                                                                           | true    |
| env:CONFIG__DEFAULT__WEB__SECURE__USE_IN_FRONTEND | environment | 1                                                                                                           | true    |
| env:CONFIG__DEFAULT__WEB__SEO__USE_REWRITES   | environment | 1                                                                                                           | true    |
+-----------------------------------------------+-------------+-------------------------------------------------------------------------------------------------------------+---------+
```

The `--level` option can be used to view only variables set at the project or environment level.

More details about a specific variable, including its properties and whether its value is inherited at the specified environment, can be viewed with `variable:get`.

```
chrisnanninga@Chriss-MBP:01:43 PM:~/SwiftOtter/Code/so-cloud-sandbox$ magento-cloud variable:get -e Integration1 env:CONFIG__DEFAULT__DEV__STATIC__SIGN
+----------------+-------------------------------------+
| Property       | Value                               |
+----------------+-------------------------------------+
| id             | env:CONFIG__DEFAULT__DEV__STATIC__SIGN |
| created_at     | 2022-11-23T15:25:35-06:00           |
| updated_at     | 2022-11-23T15:25:35-06:00           |
| name           | env:CONFIG__DEFAULT__DEV__STATIC__SIGN |
| attributes     | {  }                                |
| value          | 1                                   |
| is_json        | false                               |
| project        | 3zltyuobw3r2q                       |
| environment    | Integration1                        |
| inherited      | true                                |
| is_enabled     | true                                |
| visible_build  | true                                |
| visible_runtime | true                               |
| is_inheritable | true                                |
| is_sensitive   | false                               |
| level          | environment                         |
+----------------+-------------------------------------+
```

You can also view only a single property of a variable:

```
magento-cloud variable:get --property is_sensitive ADMIN_URL
```

The property names shown in the standard `variable:get` output can be passed to `--property`. (e.g., "is_sensitive", not "sensitive".)

When creating a new variable, you have the option of supplying the `env:` prefix as part of the name itself:

```
magento-cloud variable:create env:CONFIG__DEFAULT__GENERAL__STORE_INFORMATION__NAME --
```

... or by specifying `prefix`:

```
magento-cloud variable:create CONFIG__DEFAULT__GENERAL__STORE_INFORMATION__NAME --pref
```

(In all other commands referencing a single variable - `get`, `delete`, `update` - the `env:` prefix is simply included in the var name.)

From the table above, you can see that all properties of variables can be specified with options. You will receive interactive prompts for any properties without a default value if you do not specify the corresponding options.

The `variable:update` command works the same way to update the value or properties of a variable.

When deleting a variable with `variable:delete`, the option for declaring the project or environment level can be significant if the same variable name exists at multiple levels.

Effects on Environments

Adding or editing variables will result in a re-deployment of the affected environments(though not typically a re-build).

When the name of the variable includes the `env:` prefix, its value is available directly by name on the web server. With a variable defined for `env:COMPOSER_AUTH`, for example, you can verify the environment variable by connecting to the environment with SSH and running:

```
echo $COMPOSER_AUTH
```

By contrast, without a prefix, a value is instead only available through the `MAGENTO_CLOUD_VARIABLES` environment variable. This is a Base64 and JSON encoded value.

```
echo $MAGENTO_CLOUD_VARIABLES | base64 -d | json_pp
```

The same information can be displayed with the `ece-tools` package:

```
php vendor/bin/ece-tools env:config:show variables
```

We'll discuss more about the practical application of environment variables in a later lesson.

8.05: Describe environment Management using UI

We're now going to examine the practical aspects of managing your environments in your Cloud project. It's worth taking a moment to review what we already know about environments:

- Environments exist in a hierarchy and correspond with branches in the Git repository.
- Environments can be active, meaning they are actually deployed into working service

containers in Cloud, or inactive, meaning they exist only as a Git branch and entry in the hierarchy.
- Environments can be viewed and managed in the Project Web Interface or with the Cloud CLI tool.
- Cloud project users have varying permissions in specific environments.

We've seen that the initially provisioned environments differ between Starter and Pro plans. If the recommended structure is followed, however, the main hierarchy in either will include Production, Staging and Integration, leaving one more active environment that can be created.

Branching and merging environments (and "syncing" code) correspond with Git operations in the remote repository, as well as affecting your local environment when done with the Cloud CLI tool.

For each topic, we will look at examples using both the Project Web Interface and the Cloud CLI tool.

8.06: Demonstrate understanding of branching using UI

Branching and Merging

Cloud environments have hierarchical relationships, starting with the Master environment (equivalent to Production on Starter). Pro projects are provisioned with Production, Staging and Integration environments in a direct hierarchy; the recommended workflow for Starter is also to create a Staging environment from Master and an Integration environment from Staging.

The intent of this environment hierarchy is to facilitate good testing and deployment practices. New code changes should be pushed into the Integration environment for initial testing and, once determined stable, merged back into Staging for final testing and then to Production for deployment to the live site. (In Pro, any changes merged into Production are also expected to be immediately merged into Master.) This is a familiar version control workflow; in Cloud, the environment hierarchy applies not only to the flow of Git operations, but also to environment

variable inheritance and data syncing.

> **Reminder**
>
> For both Starter and Pro, one other active environment beyond "Production > Staging > Integration" structure is allowed. This environment can be used for whatever feature testing/prototyping needs you have.

A couple of key notes about creating new environments:

- On Pro, environments cannot be branched from Staging or Production.
- It's strongly recommended to branch new environments from Integration.

Let's take a look at the methods that can be used to branch new environments and merge changes back up the hierarchy.

Using Git

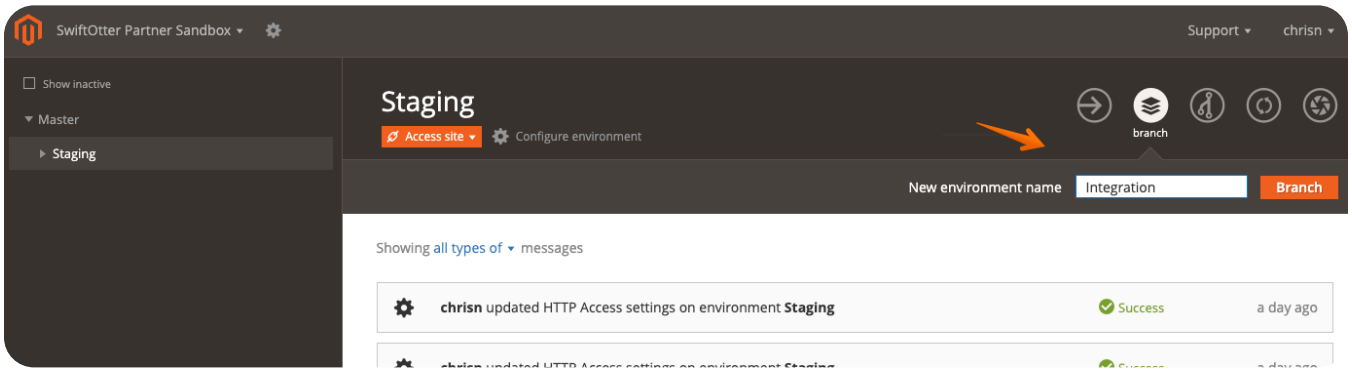
Since your Cloud project repository is a typical Git repo, you can perform any and all typical Git operations, including manually creating a branch from another. (Users must have the appropriate permissions to push such new branches to the remote repository.) If using this method for branching, there are a couple of caveats to keep in mind:

- This will not automatically provision an active environment; it will initially result only in a Git branch (i.e., an "inactive" environment). You can, however, manually "activate" such an environment, provided you have the remaining available slots.
- Regardless of where you create your branch in the Git tree, and where other branch pointers are, pushing a manually created branch will not record hierarchy in the Cloud environment; the new branch will be considered a child of Master for the purposes of data syncing, variable inheritance, and merging with the Cloud tools. For this reason, it's best to use one of the other methods for initial branching.

Merging can be done in typical Git fashion (even if the initial creation of a branch/environment was done with the Cloud tools). If a code update is merged into a branch matching an active environment and pushed, a redeployment will update the state of that environment.

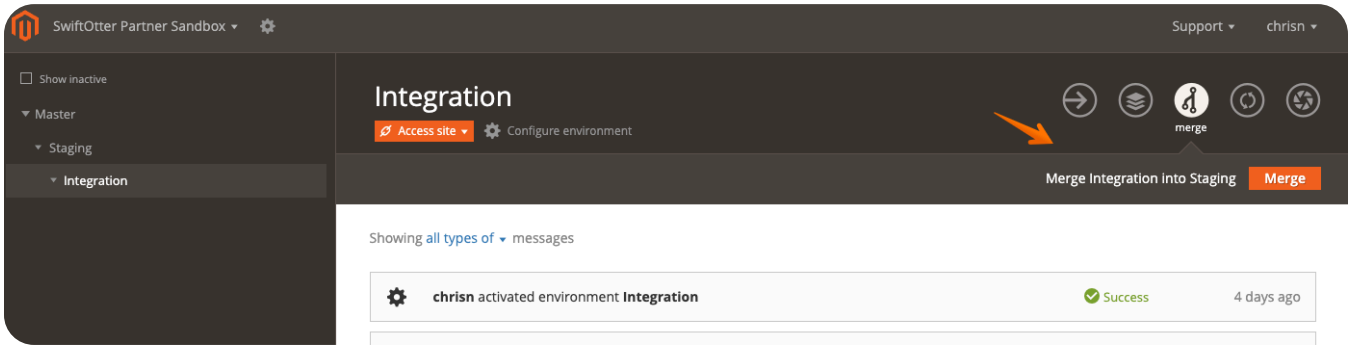
Using the Project Web Interface

When accessing an environment via the Project Web Interface, a new environment can be branched from it (provided there are enough remaining slots). This is done using the appropriate task widget as shown, providing a name for the new environment.



Creating an environment in this way will maintain its hierarchical relationship with the environment it is branched from, will result in a new Git branch in the remote repository, and will automatically kick off the provisioning of a new active environment based on the parent's files and data.

The merging process works much the same way. A merge widget is available that will kick off the merging of an environment's code state into its parent and the immediate re-deployment of that parent.



Using the Cloud CLI Tool

A set of commands in the Cloud CLI Tool support branch management.

Command	Description	Notable Arguments/Options
environment:branch (branch)	Branch an environment	`--title` - Title of the environment (can be different than ID) `--type` - Type ("staging" or "development") `--force` - Create even if branch can't be checked out locally `--no-clone-parent` - Don't clone parent env's data `--identity-file` - Path to SSH key to use
environment:checkout (checkout)	Check out an environment. Merely equivalent to a typical Git `checkout`.	`--identity-file` - Path to SSH key to use
environment:merge (merge)	Merge an environment	
environment:push (push)	Push code to an environment.	`--target` - Target branch name `--force` / `-f` - Allow non-fast-forward update `--force-with-lease` - Allow non-fast-forward if remote-tracking branch is up to date `--set-upstream` / `-u` - Set target as upstream for source

Command	Description	Notable Arguments/Options
		`--activate` - Activate environment before pushing `--parent` - (When using `activate`) Set new env parent `--type` - (When using `activate`) Type of new env ("staging" or "development") `--no-clone-parent` - (When using `activate`) Don't clone parent env's data `--identity-file` - Path to SSH key to use

> **Reminder**
>
> Remember that the Cloud CLI commands will infer the project and environment context from the local code state. If you run these commands outside a local project context, however, or wish to execute a command for a *different* environment, you can use the `--project`/`-p` and `--environment`/`-e` options.

Here's an example of branching from the currently checked out environment branch.

```
magento-cloud environment:branch NewEnvName
```

Unlike a vanilla Git branch operation, this will register the hierarchical relationship with the

parent and will provision the new active environment.

You can explicitly name the parent environment to branch from:

```
magento-cloud environment:branch NewEnvName ParentEnvName
```

The `environment:merge` command will perform the appropriate Git merge operation, push the changes to the remote repository, and re-deploy the target environment. This example merges the currently checked out branch into its parent:

```
magento-cloud environment:merge
```

The environment/branch to merge can also be explicitly named:

```
magento-cloud environment:merge NewEnvName
```

Further Reading

- Clone and Branch Management

8.07: Identify Adobe commerce Cloud Plan capabilities

The Cloud Service Plans

There are two separate subscription plans available for Adobe Commerce in the cloud: Starter and Pro. There are important differences between the two, in terms of infrastructure as well as features and service.

Below is a brief overview of significant features on each plan:

Feature	Starter	Pro
PayPal Onboarding Tool	x	x
Commerce Reporting	x	x
Continuous Integration Tools	x	x
Unlimited Users	x	x
Fastly CDN, image optimization, and WAF	x	x
New Relic APM	x	x
Platform-as-a-service (PaaS) environments	x	x
24x7 Email Support	x	x
B2B module	Paid add-on	x
New Relic Infrastructure		x
Dedicated Infrastructure-as-a-service (IaaS) for Prod/Staging: Dedicated hardware, high availability three-node setup		x
Dedicated Customer Technical Advisor		x

Commerce Reporting

Adobe Business Intelligence integrates with an Adobe Commerce instance to provide a dedicated dashboard for advanced analysis and reporting. The service integrates directly with the Commerce database to provide visualization, custom metrics, and report building, and allows for connecting other systems.

See the product details

B2B Module

The B2B (business to business) extension is a packaged suite of modules enhancing Adobe

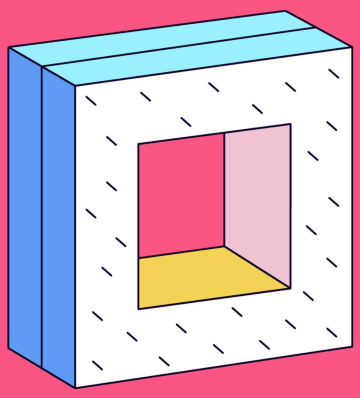
Commerce with several features. These include company accounts, specific roles and permissions for customers within a company, custom tailored catalogs for customers, quick order capabilities, requisition lists, and quotes.

The B2B extension is installed with Composer like the core Adobe Commerce packages. Access is granted as part of the Pro subscription or can be added for an additional fee for Starter.

See the product details

Further Reading

- Architecture Overview



Objective 9

Commerce Cloud CLI tool

6% of the test / 4 questions

Adobe Commerce Expert Developer Prep Guide, AD0-E716

SwiftOtter

9.01: Demonstrate understanding of updating cloud variables using CLI

To create a new variable in Cloud:

```
magento-cloud variable:create \
  --name=SOMETHING_IMPORTANT
  --value=mypassword
  --sensitive=true
```

If you don't specify a level (`--level`), you will be asked the level (`project` or `environment`) at which to create this variable. If you specify the `environment` level, you will be able to select which environment to which this applies.

To update a variable in Cloud

```
magento-cloud variable:update \
  SOMETHING_IMPORTANT
  --value=newpassword
```

This is very similar to creating a variable.

9.02: Demonstrate understanding of environment Management using CLI (CLI exclusive features :activate emails, rebase environments, snapshot, db dump, local environment setup)

Activate emails

To see if email is enabled:

```
magento-cloud environment:info
```

Check the `enable_smtp` value in the resulting table.

```
magento-cloud environment:info enable_smtp true
```

Rebase environments

Rebasing is similar to merging. Changes are brought from one branch into our current working branch. The mechanics are what makes the difference. Merging pulls the entire commit history from the branch into our working branch. Rebasing loads the changes, but smashes the commits down into one or more commits.

When synchronizing environments (branches), you have the option to rebase the changes coming into your current environment/branch.

```
magento-cloud environment:sync --rebase code
```

Further information

- Reset environment

Snapshot

Create a snapshot of the current environment. This should happen before any significant change to the environment.

```
magento-cloud snapshot:create
```

DB dump

This command pulls the database from the current environment and places it locally. You have the option to pass the `--gzip` flag to reduce the file size. You can also pass the `--environment` flag to dump a specific environment.

```
magento-cloud db:dump -f db-dump.sql
```

> This goes without saying, but you should never leave unencrypted or unsanitized databases on your development (or production) machine file systems. Ideally, you use a tool like Driver to remove personally identifiable information (PII).

Local environment setup

The Adobe Commerce Cloud environment will work with any local development environment.

> We contribute to and use Den. It allows for multiple environments, and all the services necessary to run Adobe Commerce.

If you need to troubleshoot the build process, you may want to run:

```
magento-cloud local:build
```

Further reading

- `magento-cloud` reference

9.03: Demonstrate understanding of branching using CLI

The `magento-cloud` tool has several commands to help assist with branching.

> An environment is a branch. But a branch is not always (yet) an environment.

Most of these environment-related commands have a parallel `git` command, which will make grokking these easier. However, `magento-cloud` adds to the `git` commands by also switching the Cloud environment association.

```
magento-cloud environment:branch [branch name] [parent branch]
```

Similar: `git checkout -b [branch name]`

This creates a new branch with the specified name.

```
magento-cloud environment:checkout [branch name]
```

Similar: `git checkout [branch name]`

```
magento-cloud environment:activate
```

This make the specified environment publicly browsable.

```
magento-cloud environment:merge
```

Similar: `git checkout [parent branch]; git merge [branch to merge]`

This begins the process of pushing changes back into the parent branch. For example, if we have checked out a new feature branch, we would run this command to push our updates back into the `staging` branch.

```
magento-cloud environment:sync
```

Similar: `git merge [parent branch]`

This refreshes the code (and the data, if desired) from the parent environment back into this feature branch.

```
magento-cloud environment:push
```

Similar: `git push origin [branch name]`

This pushes code from your local machine to the remote branch.

9.04: Demonstrate how to troubleshoot to cloud services? (My SQL, Redis, tunnel:info)

The first step to troubleshooting is connecting. The good news is the Commerce Cloud toolset makes this very simple.

Open the tunnel connection

Start by opening the tunnel connection.

```
magento-cloud tunnel:open
```

or you can use this for a single connection (with a little more control):

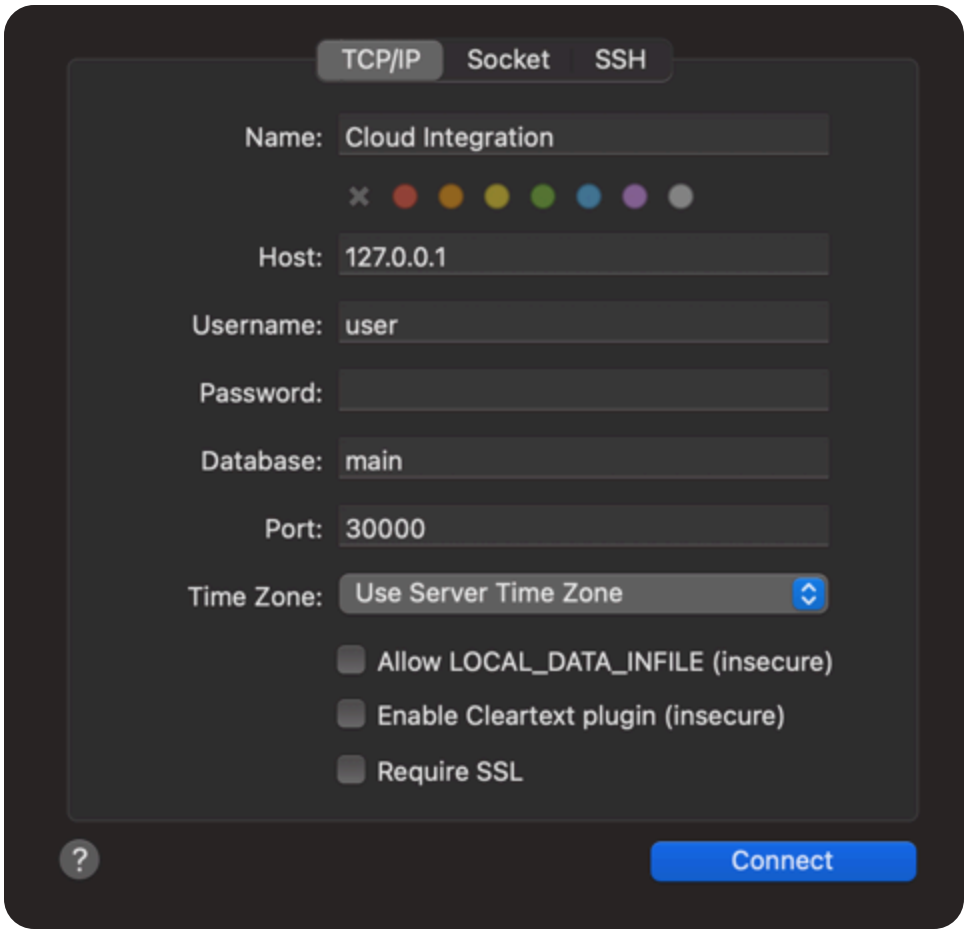
```
magento-cloud tunnel:single
```

```
➜  swiftotter-partner-sandbox git:(Integration1) magento-cloud tunnel:open
SSH tunnel opened to database at: mysql://user:@127.0.0.1:30000/main
SSH tunnel opened to redis at: redis://127.0.0.1:30001
SSH tunnel opened to opensearch at: http://127.0.0.1:30002
SSH tunnel opened to rabbitmq at: amqp://guest:guest@127.0.0.1:30003
SSH tunnel opened to redis-session at: redis://127.0.0.1:30004

Logs are written to: /Users/jmaxwell/.magento-cloud/tunnels.log

List tunnels with: magento-cloud tunnels
View tunnel details with: magento-cloud tunnel:info
Close tunnels with: magento-cloud tunnel:close

Save encoded tunnel details to the MAGENTO_CLOUD_RELATIONSHIPS variable using:
  export MAGENTO_CLOUD_RELATIONSHIPS="$(magento-cloud tunnel:info --encode)"
```

You can also list the current tunnels with:

```
magento-cloud tunnel:list
```

We use Sequel Ace for our database inspection. Using tunneling means connecting is very simple.


TCP/IP
Socket
SSH
Name:
Cloud Integration
Host:
127.0.0.1
Username:
user
Password:
Database:
main
Port:
30000
Time Zone:
Use Server Time Zone
Allow LOCAL_DATA_INFILE (insecure)
Enable Cleartext plugin (insecure)
Require SSL
?
Connect