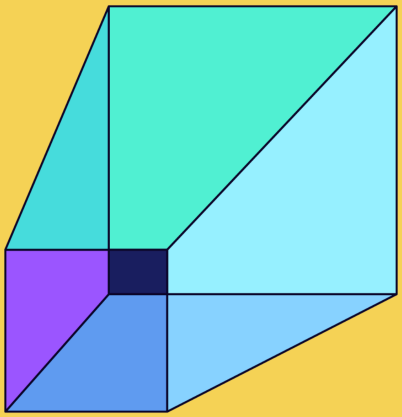


AD0-E711

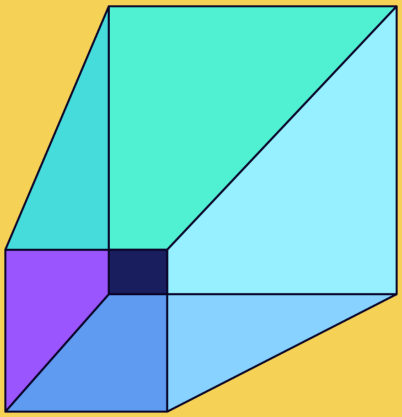
Study Guide for Professional Developer Certification

SwiftOtter

Joseph Maxwell and Vitaliy Golomoziy

Table of Contents

- 5.01 Describe cart components
- 5.02 Describe a cart promo rule
- 5.03 Given a scenario, describe basic checkout modifications
- 5.04 Given a scenario, describe basic usage of quote data
- 5.05 Given a scenario, configure the payment and shipping methods
- 5.06 Given a scenario, configure tax rules, currencies, cart, and/or checkout
- 6.01 Identify the basics of category management and products management
- 6.02 Describe product types
- 6.03 Describe price rules
- 6.04 Describe price types



Introduction

Study Guide for Professional Developer Certification, AD0-E711

SwiftOtter

Introduction

You downloaded the most comprehensive study guide for the Magento 2 Certified Professional Developer certification (testing your knowledge of Magento Open Source 2.4.3), covering the exam released in late 2021.

This test ensures that a candidate is a capable Magento 2 developer and can handle many customizations and basic new functionality requests. While I estimate this test to be a 3/10 (120 minutes, 60 questions) on the grand scale of difficulty, it is still not easy for many developers.

The goal of this guide is to provide you with knowledge and the path to obtain practical experience. The latter often comes through projects, but even then it can be misguided or not complete.

How can you achieve this certification?

I would love to say that all you have to do is read the study guide, and you will pass. That might be true for some but not the majority. To pass the test, I suggest:

- Ensure you have a development environment configured with Xdebug. I recommend XAMP (for Windows) or Valet+ for Mac. There are also Docker solutions, but I have yet to use them.
- Use an IDE like PHPStorm. Yes, this costs money, but the productivity you will gain is staggering.
- Study this guide. I know, it's not like reading a novel, but I try to make it at least somewhat interesting.
- Establish consistency in your study: such as every day from 7:30am - 8:30am, before you begin your workday.
- Obtain practical experience. There are practical experience notes throughout the study guide. Do those. Then, make up some of your own.
- Once you feel comfortable, take the SwiftOtter practice test (several takes come with this guide). It will be a reality check: "Am I ready to pass the test?"

While I have done my best in providing quality and helpful content (while also being succinct), it is your responsibility to study, practice, and prepare. With that, go and do it!

Also, would you take a moment to subscribe to our YouTube channel? I record free videos about how to become certified. You won't want to miss these!

I also wish to extend my most sincere gratitude to Vitaliy Golomoziy who has assisted in the update and development of this study guide. Isaac Phillips converted this from a Google Doc into our new platform for these study guides. And, my sister, Sarah Maxwell has reviewed it for grammatical accuracy.

All the best!

Joseph Maxwell

PS: If you have suggestions to improve this guide, please send them to me: joseph@swiftotter.com.

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—in a snap, 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.

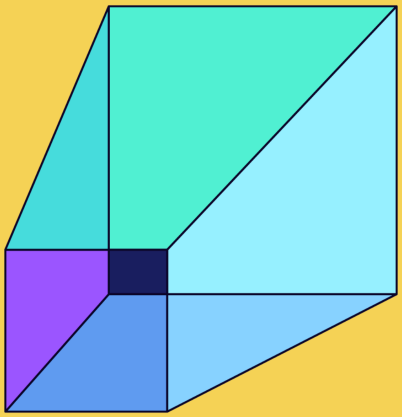
Object Mapping

The objectives that Adobe has outlined for this certification do not line up with a normal learning path. Thus, we present this study guide associated with a different structure.

Reference back to this table as you need to look up a specific subject from the test.

ADO-E711	Study Guide
1.01 Describe how the ACL works with roles and resources	2.01 Describe how the ACL works with roles and resources
1.02 Identify the components to use when creating or modifying the admin grid/form	2.02 Identify the components to use when creating or modifying the admin grid/form
1.03 Identify the files to use when creating a store/admin config and menu items	2.03 Identify the files to use when creating a store/admin config and menu items
2.01 Describe Magento file structure	1.01 Describe Magento file structure
2.02 Describe Magento CLI commands	1.06 Describe Magento CLI commands
2.03 Describe Cron Functionality	1.09 Describe Cron Functionality
2.04 Given a scenario, describe usage of the di.xml	1.03 Given a scenario, describe usage of the di.xml
2.05 Given a scenario, create controllers	1.05 Given a scenario, create controllers
2.06 Describe module structure	1.02 Describe module structure
2.07 Describe index functionality	1.07 Describe index functionality
2.08 Describe localization	1.08 Describe localization
2.09 Describe plugin, preference, event observers, and interceptors	1.04 Describe plugin, preference, event observers, and interceptors
2.10 Describe custom module routes	1.10 Describe custom module routes
2.11 Describe URL rewrites	1.11 Describe URL rewrites
2.12 Describe the Magento caching system	1.12 Describe the Magento caching system
2.13 Describe stores, websites, and store views (basic understanding)	1.13 Describe stores, websites, and store views (basic understanding)
3.01 Given a scenario, change/add/remove attribute sets and/or attributes	SAME
3.02 Describe different types of attributes	SAME
3.03 Given a scenario, use a DB schema to alter a database table	SAME

ADO-E711	Study Guide
3.04 Describe models, resource models, and collections	SAME
3.05 Describe basics of Entity Attribute Value (EAV)	SAME
4.01 Describe usage of CMS pages and blocks	SAME
4.02 Given a scenario, modify layout	SAME
4.03 Given a scenario, modify page style	SAME
4.04 Describe theme structure	SAME
4.05 Given a scenario, work with JavaScript files (basic)	SAME
4.06 Describe front-end usage of customer data	SAME
5.01 Describe cart components	SAME
5.02 Describe a cart promo rule	SAME
5.03 Given a scenario, describe basic checkout modifications	SAME
5.04 Given a scenario, describe basic usage of quote data	SAME
5.05 Given a scenario, configure the payment and shipping methods	SAME
5.06 Given a scenario, configure tax rules, currencies, cart, and/or checkout	SAME
6.01 Identify the basics of category management and products management	SAME
6.02 Describe product types	SAME
6.03 Describe price rules	SAME
6.04 Describe price types	SAME



Architecture

The basics of Adobe Commerce.

Study Guide for Professional Developer Certification, AD0-E711

SwiftOtter

1.01 Describe Magento file structure

Magento Objective: 2.01 Describe Magento file structure

Describe the Magento directory structure. What are the naming conventions, and how are namespaces established?

The root Magento directory contains these primary folders:

- `app/code` : where your custom modules are found. Hopefully, very few third-party modules are installed here because it is difficult to remember to keep these up-to-date. Installing modules through Composer provides the module vendor with an easier way to automatically update their modules and dependencies.
- `app/design/[frontend or adminhtml]` : this is where themes are stored.
- `app/i18n` : this is where translation (language) packages are stored.
- `app/etc/` : this is where the configuration files are stored:
    - `app/etc/env.php` : this is the configuration file that is per environment. This contains your database, redis, and queueing (Advanced Message Queuing Protocol or AMQP) configuration, among other things. This file should NOT be in your Git repository as it contains sensitive information.
    - `app/etc/config.php` : this file should be committed to your Git repository. It ultimately merges with the `env.php` file. This holds the configuration for enabling/disabling modules and can include defaults for Store Configuration and theming.
- `bin/` : you will type the command `bin/magento` thousands of times in your tenure as a Magento developer ... so this is where this file is stored.

- `dev/` : this contains the configuration for built-in tools, such as Grunt. It also contains Magento's test suite.
- `generated/` : files in this folder are created by Magento. These files should be built in the deployment process (ideally using a CI/CD system). This is where Magento stores factory classes, interceptor classes (used to make plugins work), proxy classes (used to lazy-load methods to prevent the constructor from triggering too early and causing a problem elsewhere in the system), and extension attribute interfaces and classes.
- `lib/` : this folder contains the internal libraries on which Magento relies. Note that jQuery is found in `lib/web/jquery/` .
- `pub/` : this is the directory that should be HTTP root (i.e., the directory that is browsed when you go to https://swiftotter.com) for the webserver. While the root Magento folder would work, you risk exposing folders such as your `var` folder with a misconfiguration. When you browse to a Magento website, the webserver first checks to see if that exact path exists as a file. If it doesn't, the `.htaccess` or Nginx configuration rewrites the request to `pub/index.php` , which begins the normal Magento request.
    - `pub/media/` : this is where the website's images are stored.
    - `pub/static/` : this contains the `.css` , `.js` and `.html` files that have been processed and are ready for the end-user to download. If a path is requested but doesn't exist as a file in this folder, Magento tries to create it when not in production mode (or if you add a store with a new language). If your website is in `developer` mode, and you run `bin/magento dev:source-theme:deploy` , most files in this directory will be symlinked to their sources throughout the Magento codebase. In production mode, you run `bin/magento setup:static-content:deploy` to process and copy files to this directory.

What's the difference? `dev:source-theme:deploy` creates symlinks from module's LESS files into the `pub/static` directory. This allows LESS files changes, in modules, to always be of the latest version.
`setup:static-content:deploy` copies all necessary files (JS, HTML and LESS compiled to CSS) to the `pub/static` directory. Once this command is run, changes to your LESS and JS files in your module will not be visible on the frontend.

Never delete the .htaccess in this directory—doing so will make your theme files not download.
\

- `setup/` : this directory contains important files related to installing Magento. If your webserver is configured so that the document root matches the Magento root directory (ie, not `/pub`), you can browse to your website `/setup` URL. This configuration should be very temporary, and production's document root should always be `/pub` . Why? Because a simple misconfiguration (face it, this can happen) can expose sensitive information in the `var/` or other directories. Magento provides an easy way to avoid this issue by ensuring a correct document root.

  But what if I need to install Magento? Either use the `/setup` URL (by changing your application's document root) temporarily. Or, better yet, use the CLI command: `bin/magento setup:install` .

- `var/` : this directory contains files that are temporarily used by the Magento application or results from using it. For example, logs are in `var/log/` . Error reports are in `var/report` . If you are using the file-system cache, this is stored in `var/cache` and `var/page_cache` . You should only use file-system caches in your development

environment and **never** on production. The reason is that file-system storage prevents horizontal scaling (scaling Magento to multiple stand-alone instances)

Do not store any important information in this directory. It can be deleted at any time.

- `vendor/` : this is where all of the Composer-installed modules are located. This directory can be deleted and re-created at any time on your development machine (doing this in production would result in your store being down). To create this directory, run `composer install` (this command is different than `composer update` in that `install` doesn't change any versions and only installs what is specified in `composer.lock`).

  Because of this, it goes without saying that code in the `vendor/` directory should never be modified. Doing so will be temporary and these changes will be destroyed the next time you run `composer update` or `composer install` .

How can you identify the files responsible for some functionality?

This is one of the first questions that a new developer asks: "How do I find this?" Here are some suggestions.

- If possible, start by finding the template that renders this information. Find a class name on that element that seems unique. Search the entire Magento root for this string. That should take you to a template. Once you find the template, search for that template's file name. This should bring you to a Layout XML file (or possibly a PHP file). In that Layout XML file, you should see the block or view model. The Layout XML file will also tell you what handle is being used and you can find the controller from that (more on this later).

- You can also set a breakpoint (with a debugger tool such as xdebug) on a line in this template. Refresh the page, or complete the action again. Once the breakpoint

triggers, you can review the call stack to find relevant code. Set breakpoints through the call stack. Cancel the request and repeat it. Step through each breakpoint you just set to get more context on this problem.

- The above doesn't always work in every situation. You can enable Block Hints with the `bin/magento dev:template-hints:enable`. This only works in Developer mode.

- As you grow in understanding of Magento, you will find that autocomplete is one of the best inventions since sliced bread. For example, if you need to load all data in an order, you know that repositories are the preferred interface for loading data. You would then add to your constructor (and autocomplete) an `OrderRepositoryInterface`. We will talk later about why you should use an `Interface` whenever possible instead of just the class itself.

Further reading:

- Template Path Hints

1.02 Describe module structure

Magento Objective: 2.06 Describe module structure

Describe module architecture.

One of the great benefits of Magento is its customization. If you are talking with a merchant who is interested in using Magento, and they ask, "Is this possible?" The answer is, "Yes." The qualifier is "how much." With many other (SaaS) platforms, there are cases in which the answer is "No."

Thus, in this guide, we will discuss how to modify Magento's default functionality to bring it in line with merchant requirements—as each will want something different.

These customizations are found in structures called "modules." Each is a self-contained unit of

functionality to perform a specific set of updates or add new functionality (both can be in the same module).

Question: how do I know when to create a new module? What happens in the module should determine its name:

- If you are adding a new feature that will involve say, over 250 lines of code (roughly speaking), create a new module for that. Another way to look at this is if you need to customize a different module in Magento, create a module for that.
- If you are creating a new child theme and need supporting view models create a new module, like `MerchantName/Theme`. This module would contain the ViewModels, LESS, and Layout XML that will be used in theme files (stored in the `app/design/frontend` directory). This module provides the supporting framework for the new theme.
- If you are creating general customizations for a specific module in Magento, I often create a module for a merchant like, `MerchantName/Catalog`. You can group customizations for areas of functionality into a Catalog module, even if the Magento modules you modify aren't just the `Magento\Catalog` module.
- If you are modifying the functionality of a third-party module, create a new module like `MerchantName/VendorModuleName`. For example, if the merchant is "SwiftOtter" and I want to customize the "Algolia" module, I would use the module name `SwiftOtter/Algolia`.

Note: that a module's name is different from its path (technically, it could be completely different). The name is what Magento uses to recognize/identify a module: `SwiftOtter_ImportTool`. The PSR-4 namespace path, though, will be `SwiftOtter\ImportTool`. For our demonstration, this module will be found in `app/code/SwiftOtter/ImportTool`, or found in `vendor/swiftotter/module-import-tool` if installed using Composer.

A Magento 2 module is found in `app/code/MerchantName/ModuleName`. In this directory, there

are several required/mandatory files:

- registration.php: Both Composer and Magento use this file. Composer determines which files to autoload (loading automatically based on their namespace path). Magento uses this to understand the modules that are available for use (see `\Magento\Framework\App\Utility\Files`).

- composer.json: Depending on how the module will be installed, this may or may not be necessary. I always include it in my modules as this will likely be required in the future. This file is primarily used by Composer (but Magento is beginning to use it too). It tells Composer about requirements and paths.

    - The `name` is the name of the module. If a module is found in the `app/code/SwiftOtter/DoSomething` directory, I would set the Composer module's name to be `swiftotter/module-dosomething`.

    - `require` specifies what other modules are needed and what PHP version to require. Most often, for custom modules, this is blank.

    - The `type` is `magento2-module`.

    - The `autoload` section specifies the `registration.php` file and the `psr-4` path to the root namespace. If your module is found in `app/code/SwiftOtter/DoSomething`, you would use this value: `SwiftOtter\\DoSomething\\` (the double backslashes are for escaping the subsequent backslash). This tells Composer the path in which to find your module, and how that is mapped to the current directory.

      A good example is if this module is included through Composer. Your file path would be `vendor/swiftotter/module-dosomething`. However, if you are using files in this module, you would use

`SwiftOtter\DoSomething\Model\ImportData`. As such, somehow `swiftotter/module-dosomething` is mapped to `SwiftOtter\DoSomething`. That is with the `psr-4` configuration.

    - Note that your module will mostly work without a `composer.json` file at this time. However, you will run into an error when you try to use the Magento CLI to uninstall a module.

- etc/module.xml: This file will likely become deprecated in favor of `composer.json`. In this file, we specify the module's name (which must match the name specified in `registration.php`). You also specify any modules on which this module depends with the `sequence` element. For example, in our `ImportTool` module, we would depend on the `Magento_Sales` module. This means that Magento will not allow the `Magento_Sales` module to be disabled IF our `SwiftOtter_ExportOrders` module is enabled:

```
<!-- app/code/SwiftOtter/ImportTool/etc/module.xml →
<?xml version="1.0"?>
<config xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:noNamespaceSchemaLocation="urn:magento:framework:Module/etc/module.xsd">
    <module name="SwiftOtter_ExportOrders" setup_version="1.0.0">
        <sequence>
            <module name="Magento_Sales" />
        </sequence>
    </module>
</config>
```

Note: you may have seen that the links in the Magento GitHub repository are to files in the `app/code/Magento` directory. Yet, you will notice in your installation with Composer, that they now reside in the `vendor/magento` directory. That is because Magento's build process

converts these modules into Composer-ready modules. You don't have to worry about any of this, other than the path to the files is different. All files inside these directories will be the same. For example:

- `app/code/Magento/Catalog` = `vendor/magento/module-catalog`
- `app/code/Magento/Sales` = `vendor/magento/module-sales`

Further reading:

- Create a New Module
- Name Your Component
- The composer.json File

What are the significant steps to add a new module?

- Create a new directory: `app/code/SwiftOtter/ExportOrders`
- Create and populate the `registration.php`, `composer.json` and `etc/module.xml`.
- Ensure that the `SwiftOtter_ExportOrders` module name is set in `registration.php` and `etc/module.xml`.
- Set the correct PSR-4 path and module name in `composer.json`.
- Run `bin/magento module:enable SwiftOtter_ExportOrders`
- Run `bin/magento setup:upgrade --keep-generated` to install this module into the database (see the `setup_module` table) and run any setup scripts.

  Note that you don't have to use the `--keep-generated` flag in your development environment. If you take this off, the `generated/` directory will be deleted. This directory stores files that are generated by Magento and by deleting them, these files

will have to be re-generated. Most of the time you can use this flag. The times where you would avoid using this flag would be if the new module includes modifications to generated code, such as extension attributes or plugins. I use this all the time. If I find that a plugin is not firing or an extension attribute doesn't exist, then I manually delete the `generated/` directory.

What are the different Composer package types?

A Composer package type is found in a module's `composer.json` file under the `type` node. Every self-contained unit of functionality (normally called a module) should have one, even though it isn't required. It is required if you intend to distribute through Packagist.org or the Magento Marketplace.

This is specifically for modules that are downloaded from the Magento Marketplace.

- `magento2-module` : if you add a module to `app/code` , or this package updates or adds to Magento functionality, this would be your choice.
- `magento2-theme` : if you create a module in `app/design/[frontend or adminhtml]` , you would use this. This is a theme.
- `magento2-language` : this would be a translation package. You can create an English (default) to Hindi package which would specify translations for strings as specified in Magento.
- `magento2-component` : this contains files that are located in the root directory. The `vendor/magento/magento2-base` directory is an example of this.
- `magento2-library` : this is the type for core framework found in `vendor/magento/framework` .

Further reading:

- Package a Component

When would you place a module in the `app/code` folder versus another location?

While there is some debate on this, for the sake of simplicity, you should place your code in the `app/code` directory.

The other common location is `vendor/` where modules specified in the project's `composer.json` file are installed.

1.03 Given a scenario, describe usage of the di.xml

Magento Objective: 2.04 Given a scenario, describe usage of the di.xml

Demonstrate the ability to use the dependency injection concept in Magento development.

Dependency injection is the idea that a class is given the classes that it needs in the `__construct` method.

In my prior to Magento 2 years, I would write a class to do something (I still write classes to do things). Let's say that I was updating some information on the order. If we needed a value from somewhere, I would fetch it in the method:

```
public function updateOrderValues($order)
{
    $customer = Mage::getModel('customer/entity')->load($order->getCustomerId());
    // ...
}
```

Here I am calling the `Mage` god class with static methods, as seen in Magento 1. This class was the single source of truth for almost anything in Magento 1.

Here were some problems:

- It was difficult to tell what my class was doing. There were `Mage::` calls thrown in here and there all over the place.
- It reduced the need to segregate code into different classes. I had no metric, other than lines of code, for what was too much.
- It made testing difficult. You can "mock" (create a "dummy" version that overrides methods to return a value you specify instead of the real value that would normally be returned) the `Mage::` class. Yet, this was difficult as you would have to do a lot of "mocking" to build a test.

The solution is dependency injection. Specifically, Magento uses constructor dependency injection.

In your `__construct` method, you list classes that you want to be injected and Magento's DI system will do that for you.

```
public function __construct(
    \Magento\Framework\App\Config\ScopeConfigInterface $scopeConfig
) {
    $this->scopeConfig = $scopeConfig;
}
```

Wow, that's cool! Now we can write methods in this class to get store configuration values.

But . . . what starts this chain of injections? DI can represent the opposite way to look at architecture: instead of your class taking the initiative to *go out and call static classes or initialize new classes*, your class simply states what classes it needs and Magento *gives your classes what it wants*.

The value is that now you can have total control over what is shipped to your class.

- You can substitute classes with `preferences`.
- You can create "new" classes with `virtualTypes`.

- You can change constructor arguments out with the `type` element in `etc/di.xml`.
- You can easily inject mocked classes for your unit tests.
- You can see exactly what other classes your class needs (dependencies).

Remember: another class always calls your class.

If you want proof of this, set a breakpoint in `pub/index.php`. This is the starting point that spins up your Magento instance. The Magento application loads up the DI container. This loads up configuration and a controller. Layout XML is initialized which then renders blocks which render templates which use view models. Through this process, events are triggered and observers listen. Models are loaded from the database.

Ok, so it makes sense that a controller is called. We can look at this as a primary touchpoint: something that the core Magento application calls. But, in working to make readable classes, we should be creating classes that function as an “assistant” by providing common functionality that can be useful elsewhere. Remember, each of these classes should do one thing. We use these classes by injecting them into the primary touchpoints.

For example:

```php
<?php
declare(strict_types=1);

// app/code/SwiftOtter/OrderExport/Controller/View/System.php

namespace SwiftOtter\OrderExport\Controller\View;

use Magento\Framework\App\Action\Context;
use Magento\Framework\View\Result\PageFactory;
use SwiftOtter\OrderExport\Action\RetrieveApiValue;

class System extends \Magento\Framework\App\Action\Action
{
    private $actionClass;

    public function __construct(
        Context $context,
        RetrieveApiValue $actionClass
    ) {
        parent::__construct($context);

        $this->actionClass = $actionClass;
    }

    public function execute()
    {
        $this->actionClass>getValue();
        // do something else
    }
}
```

In this case, Magento provides an instance of `RetrieveApiValue` to the above Controller. `RetrieveApiValue` can request an instance of another class, which can continue the chain.

Side note: while it seems verbose, I have found creating classes with one public method (`execute` , for example) are quite helpful: *1)* naming is clear as this class does one thing (this makes maintenance of this module easier), *2)* unit testing is less effort as the class is likely not massive.

Magento defaults to a single, stored instance of an injected class.

In the above example, `MyHelperClass` is created and then saved. If another class requests `MyHelperClass` , the same instance will be provided to both requesting classes.

A **factory** is an automatically-generated class (stored in the `generated/`) directory that creates an instance of another class. To request a factory, you append `Factory` to the end of a class:

```
use SwiftOtter\OrderExport\Model\MyHelperClassFactory;
```

Magento will automatically generate this class for you. You can see plenty of examples in the `generated/` directory by finding files with the word factory in their name.

Each factory (one per class) has one method: `create()` . Of course, the factory classes can be shared. But the result of the `create()` method is a unique object.

If you are creating a custom interface to create products (say, from an externally-hosted file), you would need to create multiple `Product` objects. This technique is the same whether you need to create one of these objects or multiple.

```php
use Magento\Catalog\Model\ProductFactory;

private $productFactory;

public function __construct(ProductFactory $productFactory)
{
    $this->productFactory = $productFactory;
}

public function createProducts(array $csvProductDetails)
{
    foreach ($csvProductDetails as $sku => $details) {
        $product = $this->productFactory->create();
        // … continue save process
    }
}
```

But how do I know whether to inject a class or its factory?

The delineation for using a factory is if the class "has state." In other words, is there data that is set just in this class? For example, a product object represents one product. This product has a name and price. If our constructor requests an instance of `\Magento\Catalog\Model\Product`, we are loading this object from the central object pool or repository.

In the above example, if we drop `Factory` and the `->create()` methods, we get the same object type (`\Magento\Catalog\Model\Product`), but there is a very important difference. We don't get a new instance of this `Product` class. This is one object, and it comes from the central repository.

Because the product object "has state" (a unique name, price, etc.), we need to create unique objects for each product. Without a new instance of the `Product` class, we can still call

`setName()`, and even save it. However, by using `Factory->create()`, we can create an infinite number of unique products without reusing the current instance of `Product`. The `foreach` example makes this very apparent, but even without a `foreach`, when creating a new product object (or instantiating any class that stores unique data to itself), we always want to use a factory.

On the other hand, a resource model has no state: it saves and loads data in other objects. Usually, an observer has no state: it acts on data provided to the `execute()` method and doesn't save anything to the class.

Uses a factory:

- Data models that are an interface for using data that was stored (or will be stored) in the database. This would be a product, order, customer, or a custom model you create.
- Collections for data models as filters are applied per class. This would be the product collection, order collection, or customer collection.

A factory instantiates a class. Ultimately, this represents another way to get information to an object. You can call `setter` methods and pass in dependencies. Or, you can create a new class and send dependencies to the constructor (you may or may not have setter methods). The former represents a reusable container, which can cause problems if you don't properly erase information. The latter represents a container that is thrown away (or saved to the database). All roads lead to Rome:

```
// app/code/SwiftOtter/OrderExport/Model/Action/CleanProductDetails.php
namespace SwiftOtter\OrderExport\Model\Action;

class CleanProductDetails
{
    private $product;

    public function __construct(ProductInterface $product)
    {
        $this->product = $product;
    }

    public function clean()
    {
        $this->product->setCustomAttribute('cleaned', true);
    }
}
```

Now, here is how we would use this class:

```
private $cleanProductDetailsFactory;

public function __construct(
    CleanProductDetailsFactory $cleanProductDetailsFactory
) {
    $this->cleanProductDetailsFactory = $cleanProductDetailsFactory;
}

public function methodThatNeedsACleanedProduct(Product $product)
{
    $cleaner = $this->createProductDetailsFactory->create(['product' => $product]);
    $cleaner->clean();

    return $product;
}
```

Notice how we are creating this class and passing a product into the constructor. In this example, we could refactor our `CleanProductDetails` to remove the product from the constructor and instead pass it to the `clean()` method. Which way is better? I like to use the constructor injection when I find myself passing an object from method to method in a class. If `clean()` had four parameters and those four parameters are passed to method after method, it is much cleaner (pun intended) to use constructor injection.

Further reading:

- Dependancy Injection

How are objects realized in code?

When using the Object Manager, we must remember a few things:

- There are few, if any cases where we should actually type the words `ObjectManager` in our code. One instance is in a Factory class (that we create to simplify the process

of creating an object). Another is to instantiate an object where the code loads or assembles the class path and cannot be specified in the constructor. Often, though, we can use `di.xml` to realize this (see `Magento\Catalog\Model\ProductLink\CollectionProvider`).

- Object Manager should never be found in a `.phtml` template. This is a code smell and is an immediate flag that the developer does not know what they are doing. Instead, use view models to locate and enhance functionality. This takes a few extra minutes but will save time in the long run (View Models).
- We should never type `new \SwiftOtter\ExportOrders\Model\Data();` or something similar. Why? Because the Object Manager handles this. This enables us to harness the power and flexibility where classes can be loaded or created just for our model (not to mention the tools available in `di.xml`). In addition, manually creating said class paths makes our code brittle and difficult to test—we cannot use mocks.

I get this, but **how are objects realized in code?** Great question.

The original idea is when a class is requested in a constructor, Magento parses the parameter types. If a type is already loaded in its array of already-instantiated objects, this object is used. If a type has not been loaded, the type is created. The type isn't just created, but the above process is also executed for that requested type, as it may have constructor arguments.

Here's an example. I'll create a new controller (note that I am not using class path aliases for the sake of example):

```
public function __construct(
    \SwiftOtter\OrderExport\Model\Exporter $exporter
    \Magento\Framework\App\Action\Context $context
) {
    // ...
}
```

Practical experience #1:

- Open `vendor/magento/framework/App/ObjectManager/Environment/Developer.php` (when in Developer mode, per `app/etc/env.php`'s `MAGE_MODE` configuration) or `vendor/magento/framework/App/ObjectManager/Environment/Compiled.php`.
- Set a breakpoint in the `configureObjectManager` method.
- Navigate to the home page in your browser.
- Step into the first `get` method.

Practical experience #2:

- Open `vendor/magento/module-cms/Controller/Index/Index.php`.
- Set a breakpoint (see appendix) on the class definition (where it reads `class Index extends Action ...`. This breakpoint is triggered when this class is loaded: see the `includeFile` method in the call stack).
- You will have to step back up to the `Base` router: `Magento/Framework/App/Router/Base.php`.
- Step into `$this->actionFactory->create($actionClassName);`.
- You are now able to step into and through the loading process for this controller's dependencies.

For the forthcoming discussion, please work through the above practical experience AND read the below.

When the correct URL is triggered for this controller and Magento needs to instantiate this class, here is what happens:

- `ObjectManager::create($className)` is called (\Magento\Framework\ObjectManager\ObjectManager). Note that you can also call

`ObjectManager::get($className)`. The difference is that `create` always returns a new instance of the requested class. `get` returns an instance: this may be one that has already been instantiated.

- The `ObjectManager` checks to see if a `preference` exists (\Magento\Framework\ObjectManager\Config\Config) for the requested class. This is how your constructor can depend on an `Interface` (which cannot be instantiated), yet your class can load up just fine: if your `etc/di.xml` file has a `preference` specified, for example:

```
public function __construct(
    \Magento\Catalog\Api\ProductRepositoryInterface $productRepository
) {
  // ...
}
```

```
// etc/di.xml
<preference for="Magento\Catalog\Api\ProductRepositoryInterface"
                    type="Magento\Catalog\Model\ProductRepository" />
```

In the above, when you request a `ProductRepositoryInterface`, the first thing Magento does is see if there is a `preference` for this type. If so, it uses the class specified in the `type`. If not, it tries to instantiate the class. Of course, we know that PHP won't even try to instantiate an `Interface` and you will experience a fatal error.

- The Magento mode configuration determines which file handles the creation of the object. Developer (`\Magento\Framework\ObjectManager\Factory\Dynamic\Developer`) is more verbose and throws errors easier. This allows you to catch problems sooner. Production (`\Magento\Framework\ObjectManager\Factory\Dynamic\Production`)

allows the process to continue, even if circular dependencies exist—which, by the way, would bring the website to a grinding halt.

- The next step is to get the instance type. We already covered how we can change which class is initialized with a `preference`. `\Magento\Framework\ObjectManager\Config\Config::getInstanceType()` looks to see if this class is a virtual type. If the request class is indeed a virtual type (di.xml - Virtual Types), Magento utilizes the `type` attribute specified in the `virtualType` XML node (remember that a virtual type is a instance of class specified in the `type` parameter, but with changed constructor arguments).
    - Once the type is determined, we check if there are plugins for this class. If plugins are present and valid, `\Interceptor` is appended to the class path. If you request, `\SwiftOtter\OrderExport\Model\OrderData` AND there is an active plugin, the type returned will be `\SwiftOtter\OrderExport\Model\OrderData\Interceptor`. The `Interceptor` class is auto-generated and is stored in the aptly-name `generated` directory, following the class path (like `generated/SwiftOtter/OrderExport/Model/OrderData/Interceptor.php`). This generated class matches the functions specified in the parent class and triggers any plugins when your method is called.
- The parameters are loaded from the `__construct()` method (`\Magento\Framework\ObjectManager\Definition\Runtime` and `\Magento\Framework\Code\Reader\ClassReader`).
    - The arguments are then “resolved.” Meaning, Magento will follow the above process for each class in the entire tree (this class’ constructor requires this class, which requires this class—you get the point).
    - This work happens in

`\Magento\Framework\ObjectManager\Factory\AbstractFactory::resolveArgument()`. I highly recommend you step through this.

- As seen in this code, you can specify a `shared` option to control whether or not an object is loaded from the central object repository or simply created. Where do we set this? In `etc/di.xml`, like:

```
<!-- etc/di.xml -->
<type name="SwiftOtter\OrderExport\Model\OrderData">
    <arguments>
        <argument name="Product" type="xsi:object" shared="false">
            Magento\Catalog\Model\Product
        </argument>
    </arguments>
</type>
```

The above will inject a fresh, brand-new `Product` object into your class. This would work if your class only needs one `Product` object. If you need more, you would want to utilize a `Factory`, like `ProductFactory`.

I have never used `shared` before, and instead opt for consistency and using factories.

- The object is created with the `createObject()` method. This is a very simple method with this code:

```
return new $type(...array_values($args));
```

Practical experience: build out the interface and implementation model classes.

Why is it important to have a centralized object creation process?

This allows tremendous control over how objects are instantiated:

- Preferences to change which class is loaded.
- Virtual types to create a new “class” with the same code as another class: the difference is that you can change which values (objects) are sent to the constructor.
- Plugins to affect functionality of literally any public method in the system (including `abstract` methods).
    - Plugins are not available for `private` or `protected` (which Magento is working to remove) methods.

Identify how to use DI configuration files for customizing Magento.

In this section, we will focus on what it takes to customize values sent to a particular class. This happens through the `type` node in `etc/di.xml`. For example, let’s create a class:

```
namespace SwiftOtter\OrderExport\Model;

class OrderDataRetriever
{
    private $customerSession;

    private $sort;

    private $filterModels;

    public function __construct(
        string $sort,
        array $filterModels,
        \Magento\Customer\Model\Session $customerSession
    ) {
        $this->customerSession = $customerSession;
        $this->sort = $sort;
        $this->filterModels = $filterModels;
    }
}
```

And, now, let's change every single argument value.

```
<!-- etc/di.xml -->
<!-- … -->
<type name="SwiftOtter\OrderExport\Model\OrderDataRetriever">
    <arguments>
        <argument name="sort" xsi:type="string">created_at ASC</argument>
        <argument name="filterModels" xsi:type="array">
            <item name="customerIdFilter" xsi:type="object">
              SwiftOtter\OrderExport\Filter\CustomerId
            </item>
            <item name="totalFilter" xsi:type="object">
              SwiftOtter\OrderExport\Filter\TotalFilter
            </item>
        </argument>
        <argument name="customerSession" xsi:type="object">
          Magento\Customer\Model\Session\Proxy
        </argument>
    </arguments>
</type>
```

There are a number of argument types available (di.xml - Argument Types). Above, I demonstrate three uses:

- `string`, `boolean` or `number`: your value is hard-coded into the `di.xml` file. Note that you can place a `di.xml` file in the `adminhtml` or `frontend` directories (like `etc/adminhtml/di.xml`) and override the value specified in `etc/di.xml`.
- `array`: this generates an array of values.
- `object`: this is used to set or change input types. Note that if the `__construct` method specifies a class, you must still obey that and the new class injected must extend or implement this class. You can also use this to inject a virtual type (see the next section).
- `const`: sends in a value as specified by a class constant.

Further reading:

- DevDocs - The di.xml file
- Alan Storm - Shared Instances and Dependency Injection
- Firebear Studio - Dependency Injection in Magento 2
- Webkul - Proxy Design Pattern And Code Generation

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

Overriding a native class: use a `preference`. Use this sparingly **unless** you are setting a preference for an interface (preferring a `Product` class when `ProductInterface` is set in the constructor). If you need to change how core functionality works, do everything you can to use an alternative method such as plugins. There are some cases where you must adjust a `private` or `protected` method—only in this case you should use a `preference`.

Inject your class into another object. Instead of using a global `preference` to update some core functionality, you can use the `type` and `argument` nodes to inject your updated version of a class into the class that originally requested the faulty core class. You would do this if you need to use surgical precision to change core functionality: class A depends on class B and we need to modify class B. Because our changes are not necessary anywhere else, we use a `type` and `argument` configuration on class A to carefully substitute our class for class B.

What about virtual types? Practically speaking, I use these very little. The use case is if we have a given class, but want to change which dependencies are passed to it, then we have a use case for a virtual type. The other thing that can be helpful is for widgets. Injecting long class names is awkward, so you can create a `virtualType` to shorten the class name. One thing to remember that is very confusing about virtual types: when debugging a virtual type, the class will be the original type for the virtual type. This is because Magento doesn't generate anything for a virtual type. Instead, it instantiates the original type but with a new set of constructor arguments—that's it.

Practical experience: create an event observer for the `catalog_product_load_after` event. Inject an instance of the `\Magento\Tax\Model\ResourceModel\TaxClass\CollectionFactory` and load the tax class for the loaded product.

Given a scenario, determine how to obtain an object using the ObjectManager object. How would you obtain a class instance from different places in the code?

Please see the example code and demonstrations for this.

Always use the `__construct` method to obtain dependencies. If you are working with a `.phtml` template, use a ViewModel (which implements `\Magento\Framework\View\Element\Block\ArgumentInterface`).

Example:

```
<!--
  app/code/SwiftOtter/OrderExport/view/adminhtml/layout/sales_order_view.xml -->
<?xml version="1.0"?>
<page xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:noNamespaceSchemaLocation="urn:magento:framework:View/Layout/etc/page_configur
    <body>
        <referenceBlock name="content">
            <block name="order.controls">
                <arguments>
                    <argument name="view_model" xsi:type="object">
                      SwiftOtter\OrderExport\ViewModel\OrderControls
                    </argument>
                </arguments>
            </block>
        </referenceBlock>
    </body>
</page>
```

Then, in your template:

```
<?php $viewModel = $block->getData('view_model'); ?>
```

Further reading:

- View Models

1.04 Describe plugin, preference, event observers, and interceptors

Magento Objective: 2.09 Describe plugin, preference, event observers, and interceptors

Demonstrate an understanding of plugins.

Thanks to the auto-generated `Interceptor` classes, we can customize the arguments for a method (`before` plugin), whether or not the method is run (`around` plugin) and change the output from the method (`after` plugin). You are probably giddy with excitement over the power that this gives you ... but remember, with great power, comes great responsibility. Pfffffft, the air is quickly being released from our excitement. It is replaced with a sense of deep gravity.

If you need to adjust core functionality or how another module behaves, the plugin should be the first tool that we reach for in our toolbox.

When you do create a plugin for a particular module, make sure you add a dependency for your current module on the module you are updating. This prevents the website blowing up if the core module is ever disabled.

Plugins attach to `public` methods. They cannot be applied to (*this is very important to remember*):

- Final methods or classes
- Private or protected methods
- Static methods
- Constructors
- Virtual types (remember, these are the original classes, just with different constructor arguments).

`before` changes input parameters

These are returned as an array, in the order specified in the original method. Example:

```php
<?php
declare(strict_types=1);
// app/code/SwiftOtter/OrderExport/Model/OrderDataRepository.php

namespace SwiftOtter\OrderExport\Model;

use SwiftOtter\OrderExport\Model\OrderData;

class OrderDataRepository
{
    public function save(OrderData $model, bool $isFlatTable)
    {
        // this will save our OrderData
    }
}
```

And now, the plugin:

```php
<?php
declare(strict_types=1);
// app/code/SwiftOtter/OrderExport/Plugin/UpdateFlatTableArgument.php

namespace SwiftOtter\OrderExport\Plugin;

class UpdateFlatTableArgument
{
    public function beforeSave(
        \SwiftOtter\OrderExport\Model\OrderRepository $subject,
        $model,
        $isFlatTable
    ) {
        return [$model, true];
    }
}
```

Finally, the configuration for the plugin:

```xml
<?xml version="1.0"?>
<config xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
        xsi:noNamespaceSchemaLocation="urn:magento:framework:ObjectManager/etc/config.
    <type name="SwiftOtter\OrderExport\Model\OrderDataRepository">
        <plugin name="UpdateFlatTableArgument"
             type="SwiftOtter\OrderExport\Plugin\UpdateFlatTableArgument" />
    </type>
</config>
```

Flush the cache and presto! Your plugin should be triggered.

Notes:

- The plugin type is only specified in the plugin file. You do not configure this in your

plugin XML configuration.

- I typically update one method's functionality per class (before and after in the same class would be fine). Technically speaking, you can point multiple classes to a plugin, but that isn't ideal as it's easy to lose track of what's happening where.
- While plugins can be stored anywhere, standard protocol is to put them in your module's `Plugin/` directory. I suggest naming these plugins according to what they are doing. Again, this is easy to read and understand what is going on.

`around` determines whether or not the original method is executed.

Magento documentation warns that `around` plugins can cause unintended consequences, increase stack traces, and affect performance if other plugins are required to successfully complete a task. Theoretically, the only use case for around plugins is when the execution of all further plugins and original methods needs termination.

```
// around plugin for above MyRepository
public function aroundSave(
    \SwiftOtter\OrderExport\Model\OrderRepository $subject,
    callable $proceed,
    $model,
    $isFlatTable
) {
    // do we want to proceed? Maybe we return a value we load in this method
    // and skip proceeding?
    return $proceed($model, $isFlatTable);
}
```

`after` alters the output of a method.

`after` also provides the parameters that were sent to the method itself, allowing for differing alterations based on these parameters.

```
// after plugin for above MyRepository
public function afterSave(
    \SwiftOtter\OrderExport\Model\OrderRepository $subject,
    $result,
    $model,
    $isFlatTable
) {
    // do something to $result
    return $result;
}
```

Sort order

Magento provides the capability to specify a `sortOrder` for each plugin. For most plugins, it is unnecessary or even discouraged to set the `sortOrder`. However, if you need to change the input sent to another plugin (customizing a module from the Magento Marketplace), then `sortOrder` would have a valid use case.

Plugin entries in `di.xml` are executed in a lowest to highest sort order. The complicating factor is when there are multiple plugins for the same method in a class.

When there are multiple plugins for the same method in a class, here is how it works: `before` plugins are executed from lowest sort order to highest sort order, last. `after` plugins are executed from highest sort order to lowest sort order, last. Please read and study the example in DevDocs below.

Practical experience:

- Create an `after` plugin for `Magento\Payment\Model\MethodList::getAvailableMethods()`. Remember that you can use the `$quote` input variable to provide information for additional filters for

the payment methods.

Further reading:

- Component Load Order
- Plugins - Prioritizing Plugins

Demonstrate how to create a customization using an event observer. How are observers registered? How are they scoped for frontend or backend?

Create an `events.xml` file in your module's `etc/` directory. Remember that this file can be placed in an area's directory, like `etc/frontend/events.xml`. This prevents the event from being executed everywhere. For example, you may want to allow a payment method to only be used in the admin panel. You would create an observer in `etc/adminhtml/events.xml` like:

```
<?xml version="1.0"?>
<config xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
        xsi:noNamespaceSchemaLocation="urn:magento:framework:Event/etc/events.xsd">
    <event name="payment_method_is_active">
        <observer name="ConfigurePaymentIsActive"
             instance="SwiftOtter\OrderExport\Observers\ConfigurePaymentIsActive"/>
    </event>
</config>
```

On the `observer` declaration, you can disable another observer using the `disabled` attribute. This is an easy way to rewrite functionality on an observer (for example, if a third-party has an observer that is causing problems). You can also specify the `shared` attribute to control whether or not the observer is a fresh instance of the class (by default, observers are shared).

Then, you would create the observer:

```php
<?php
declare(strict_types=1);
// app/code/SwiftOtter/OrderExport/Observer/ConfigurePaymentIsActive.php

namespace SwiftOtter\OrderExport\Observer;

use Magento\Framework\Event\ObserverInterface;
use Magento\Framework\Event\Observer;

class ConfigurePaymentIsActive implements ObserverInterface
{
    public function execute(Observer $observer)
    {
        $payment = $observer->getData('method_instance');
        if ($payment->getCode === MyPaymentMethod::code) {
            $observer->getData('result')->setData('is_available', true);
        }
    }
}
```

Note that many tutorials use `getResult()` instead of `getData('result')`. Both yield the same result.

I find the place for events is becoming smaller and smaller. When using events, do not rely on it running before or after another observer. If you need to do this, use a plugin for another observer.

Practical experience:

- Set a breakpoint in `\Magento\Review\Observer\CatalogProductListCollectionAppendSummaryFieldsObserver::execute()`.

- Visit a product page.

Further reading:

- Events and Observers
- Observers Best Practices

How are automatic events created, and how should they be used?

Magento automatically triggers events for operations relating to the database (`\Magento\Framework\Model\AbstractModel`):

- Delete before and delete after
- Save before and save after
- Save commit after (once the transaction is complete)
- Load before and load after
- Clear

These methods use the model's `_eventPrefix` variable (`Magento/Theme/Model/Theme::$_eventPrefix`). In this example (where the prefix is `theme`), if you wanted to attach some functionality to saving this theme model, you could use these events:

- `theme_delete_before`
- `theme_delete_after`
- `theme_save_before`
- `theme_save_after`
- `theme_save_commit_after`
- `theme_load_before`
- `theme_load_after`
- `theme_clear`

But what if the model I want to affect has no `_eventPrefix` specified? You have three options:

- Use the default `core_abstract` as the prefix.
- In the observer, check if the value (as set by the model's `_eventObject` variable which defaults to `object`) is an `instanceof` the class you wish to modify. If so, make your changes.
- Or, use a plugin on the resource model's `save()`, `load()` or `delete()` methods. Much better.

1.05 Given a scenario, create controllers

Magento Objective: 2.05 Given a scenario, create controllers

Critical note before we get started:

Any response other than HTML from a controller is a code smell. Why? Because in our world, pretty much the only other response types are JSON and XML. There could be a plain text response, but that's few and far between.

JSON is returned from the REST API. The REST API:

- Provides structured data formats for incoming and outgoing data. You have access to object-oriented development. With a controller, you have to mangle an array and that leads to poor design practices.
- A follow-on to this is that you can use Extension Attributes to augment the data that is returned. Better yet, Extension Attributes can be associated to an ACL role. This automates the process of determining whether or not data is returned.
- Is faster than controllers (although only slightly in 2.4).
- Has authentication baked right in.

How do you identify which module/controller corresponds to a given URL? What would you

do to create a given URL?

Magento's URLs are split into three segments:

```
https://example.site/catalog/category/view
```

- `catalog`: comes from the `frontName` attribute in a `routes.xml` (`Magento/Catalog/etc/frontend/routes.xml`) (this must be inside of either the `frontend` or `adminhtml` areas). Remember, you can search the codebase for `frontName="[FIRST SEGMENT OF URL]"` to help locate the module that is powering that controller. In this case, it is the `Magento\Catalog` module.
- `category`: inside the Controller directory (you must use this directory name), you will find directories. For admin requests, first navigate into the `Adminhtml` directory (this also is a requirement). In this case, `category` is found here (`Magento/Catalog/Controller/Category/`).
- `view`: this directly maps to the file inside the category directory, `Magento/Catalog/Controller/Category/View.php`.

These are one of the few standardized paths that Magento requires for PHP classes. One thing to remember is that you will rarely, if ever, see the `catalog/category/view` or `catalog/product/view` URLs. This is because these URLs are mapped in the `url_rewrite` table FROM a friendly path (`/my-product-url-key`) TO the Magento path (`/catalog/product/view/id/12345`).

You can also append parameters to the end of this URL. Notice the `id/12345` above. After the first three parts, you can use a key/value system where the first is the key and the second is the value. These are used in your code by injecting an instance of `\Magento\Framework\App\RequestInterface` and calling `getParam()`. Note that this

injection is already found in the controller's `getContext()` method.

Important note:

Inside of a `routes.xml` file, there are two attributes to describe the route:

```
<?xml version="1.0"?>
<config xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:noNamespaceSchemaLocation="urn:magento:framework:App/etc/routes.xsd">
    <router id="standard">
        <route id="order_export_id" frontName="order_export">
            <module name="SwiftOtter_OrderExport" />
        </route>
    </router>
</config>
```

This is confusing, but the `frontName` matches the URL. However, the ID is used to formulate the layout handle. Here is an example:

```
https://example.site/order_export/directory/phpClass
```

Here is the layout handle:

```
order_export_id_directory_php_class
```

Here is the controller that is executed:

```
SwiftOtter\OrderExport\Controller\Directory\PhpClass
```

Anatomy of a controller

Controllers must implement `Magento\Framework\App\ActionInterface`. This interface

contains one method, `execute()`, which is called when this controller matches the URL path.

Magento provides more specific interfaces, for example, `Magento\Framework\App\Action\HttpGetActionInterface`. Though these interfaces do not bring any additional functionality, they can be used in the Magento system to understand the purpose of the controller's existence:

```
if ($controller instanceof HttpPostActionInterface) {
    // this must be a POST request, so do something accordingly.
}
```

A controller's constructor must require `Magento\Framework\App\Action\Context`. This class provides access to many different areas of functionality such as `getRequest()`, `getResponse()`, and `getUrl()`.

The `execute` method must return an object that implements (`\Magento\Framework\App\Http`):

- `Magento\Framework\Controller\ResultInterface`
- or `Magento\Framework\App\Response\HttpInterface`.

Response types

Implementors of `ResultInterface` (`\Magento\Framework\App\Http`). Please note that these are the return types. But, you must obtain an instance of the desired class by injecting its factory into the controller's constructor.

- Page (`\Magento\Framework\View\Result\Page`): this returns HTML loaded from a layout handle.
- Redirect (`\Magento\Framework\Controller\Result\Redirect`): this represents a 301

or 302 redirect.

- JSON (`\Magento\Framework\Controller\Result\Json`): this returns a JSON rendering. Please note that there are very few instances where it is good to use a JSON response. Instead, consider using the API (great example here: Custom API for Magento 2).
    - Some of these return types have a custom-built factory. See this (`\Magento\Framework\Controller\Result\JsonFactory`).
    - If you are returning JSON, the incoming request is almost certain to be an API request. Again, please use the API. If, for some reason you can't, you can do a basic check to ensure that the incoming request is an API request:

```
// inside a controller:
public function execute()
{
    if ($this->getRequest()->isAjax()) {
        // ...
    }
}
```

- Forward (`\Magento\Framework\Controller\Result\Forward`): this internally forwards to another controller without changing the URL.
- Raw (`\Magento\Framework\Controller\Result\Raw`): this returns whatever you want to be returned.

`HttpInterface` (`\Magento\Framework\App\Response\HttpInterface`) implementors are returning files and the web API. You will rarely, if ever, create a controller that returns an `HttpInterface` .

How do you get data from a controller to be rendered?

When you create a controller to listen to a specific URL, one of the biggest questions is how you get objects to be rendered in the response's HTML. The old way was to use a controller to

load all necessary objects. The new way is actually more simple and straightforward: use view models in your templates.

The best answer is:

- For requests that render HTML, the controller should have little to no involvement in loading data.
- For requests that redirect or return AJAX (again, consider that the API is better), the controller will likely load data.

I would like to focus on the first scenario, where we need to render HTML.

For HTML to be rendered, we need to at least have a `.phtml` template file. If that's confusing, please skip ahead to the next Objective.

We can also utilize a PHP class with a view model (again, see Objective 3). The template loads data from the view model and renders it.

Remember, there is nothing stopping us from creating multiple classes. In fact, to an extent, the more classes the better as this allows classes to be named intelligently, according to how the class will be used. Ideally, we have one or two `public` methods per class with a number of supporting `private` methods.

Here is another way to look at where data should be loaded:

- Web request initializes at index.php
- Routing occurs
- Controller's `execute` method is called
- HTML is rendered
    - This is where you should load data.
- Output is flushed and returned to browser

What about creating an additional class to act as an intermediary between the incoming

request and the view model? It would look something like:

```php
<?php
declare(strict_types=1);
// app/code/SwiftOtter/OrderExport/Model/ProductDisplayRequest.php

namespace SwiftOtter\OrderExport\Model;

use Magento\Catalog\Api\Data\ProductInterface;
use Magento\Catalog\Api\ProductRepositoryInterface;
use Magento\Framework\App\RequestInterface;
use Magento\Framework\Exception\NotFoundException;

class ProductDisplayRequest
{
    /** @var RequestInterface */
    private $request;

    /** @var ProductRepositoryInterface */
    private $productRepository;

    /** @var ProductInterface|null */
    private $product;

    public function __construct(
        RequestInterface $request,
        ProductRepositoryInterface $productRepository
    ) {
        $this->request = $request;
        $this->productRepository = $productRepository;
    }
```

```
    public function getProduct(): ProductInterface
    {
        if (!$this->product) {
            $id = (int)$this->request->getParam('id');
            if (!$id) {
                throw new NotFoundException(__('No product was specified.'));
            }
            $this->product = $this->productRepository->getById($id);
        }

        return $this->product;
    }
}
```

Now, you can inject this class into any view model (or any class in your module). Your controller's constructor is not overloaded with a large number of dependencies.

Note: `getProduct()` is guaranteed to return a `ProductInterface`. But what happens if no `id` parameter is set? As you can see, an exception is thrown. This is a way to "return" a different response. We can throw any exception that implements the PHP `Throwable` interface. For example:

```
try {
    return $this->productDisplayRequest
                    ->getProduct()
                    ->getName();
} catch (NotFoundException) {
    return __('The product was not specified in the URL');
} catch (NoSuchEntityException) {
    return __('The product requested is not found in the database');
}
```

Notes:

- If you need the `Product` in the controller, say, to set the title of the page, you can inject a class like we wrote above. With this approach, we rely on a separate class to load data instead of the controller.
- I placed this class in the `Model` directory as it is responsible for loading this data.

Your controller calls `setProduct()`. Now, anywhere in your module that you want to access this product, you inject `ProductTransport` (for example, a view model). Note that `getProduct()` *could return a* `null`*, so you must always check for that*.

We will discuss these steps more in the next objective, but let's suffice it to list the steps here:

- Create a controller. This controller does not need to load any data unless it is using this data (like the product's name) to change the pages title, or something similar.
- Developer creates a layout XML file for this controller's handle.
- Developer creates a class to load the required information.
- Developer creates a view model (stored in your module's `ViewModel` directory, see more information about this in Objective 3)

```php
<?php
declare(strict_types=1);

namespace SwiftOtter\OrderExport\ViewModel;

use Magento\Catalog\Api\Data\ProductInterface;
use Magento\Framework\Exception\NoSuchEntityException;
use Magento\Framework\Exception\NotFoundException;
use Magento\Framework\View\Element\Block\ArgumentInterface;
use SwiftOtter\OrderExport\Model\ProductDisplayRequest;

class ProductRenderer implements ArgumentInterface
{
    /** @var ProductDisplayRequest */
    private $displayRequest;

    public function __construct(ProductDisplayRequest $displayRequest)
    {
        $this->displayRequest = $displayRequest;
    }

    public function getProduct(): ?ProductInterface
    {
        try {
            return $this->getProduct();
        } catch (NotFoundException $ex) {
            return null; // invalid product ID
        } catch (NoSuchEntityException $ex) {
            return null; // no ID specified in request
        }
    }
}
```

In this example, our view model is handling exceptions, and you can just as easily move

it to the `ProductDisplayRequest` class. However, having exception handling in the `ProductDisplayRequest` means that you won't know what the specific error is. If it doesn't matter, then move it. If it does matter, this is the way to go.

- Developer configures a `block` directive in Layout XML which loads the view model as an argument and specifies a template, like:

```
<!-- excerpt from
       app/code/SwiftOtter/OrderExport/view/frontend/layout/order_export_sample.
-->
<block name="product.renderer"
  template="SwiftOtter_OrderExport::display-product.phtml">
    <arguments>
        <argument name="view_model" xsi:type="object">
           SwiftOtter\OrderExport\ViewModel\ProductRenderer
        </argument>
    </arguments>
</block>
```

- Then, in your template, for example:

```php
<?php
// app/code/SwiftOtter/OrderExport/view/adminhtml/template/display-product.phtml

/** @var \SwiftOtter\OrderExport\ViewModel\ProductRenderer $renderer **/
$renderer = $block->getData('renderer');
$product = $this->viewModel>getProduct();
?>

<?php if(!$product): ?>
    <?= __('No product was found.') ?>
<?php else: ?>
    <?= $product->getName() ?>
<?php endif; ?>
```

The above example is not found in the sample project but should serve as a blueprint for how to move data from the controller to a PHP template.

How would you create an admin controller?

To create an admin controller, you need to:

- Declare a route in your module's `etc/adminhtml/routes.xml`. This only happens once per module.
- Create a controller in your module: `Controllers/Adminhtml/[URL_SEGMENT_2]/[URL_SEGMENT_3].php`
- Configure a new ACL entry in `etc/acl.xml` (at least one per module) and use that ACL ID in the controller's `ADMIN_RESOURCE` constant.
- Unless you disable Add Secret Key to URLs (in Stores > Configuration > Admin > Security, which is NOT advised), you will not be able to type in a URL in the Magento admin. You must navigate through a link:
    - You can add a link to the new controller in `etc/adminhtml/menu.xml`, a

template, or elsewhere.

- You can generate a URL to this controller by injecting an instance of `\Magento\Backend\Model\Url` to a View Model.
- Or, in a controller, you can call `$this->getUrl()` (which ultimately calls the backend URL model).

The admin controller will extend `\Magento\Backend\App\Action`. It should implement the type of request that this controller represents, for example `HttpGetActionInterface`.

Side note: the Magento Registry (`\Magento\Framework\Registry`) is now deprecated. Please do not use it. Instead, create your own registry with specific getters and setters. You are thus able to type-hint and even perform error checking. The downside is this is more verbose. The upside is that your application is less prone to errors.

Practical experience: follow along in creating a controller that returns JSON.

How do you ensure the right level of security for a new controller?

Set the `ADMIN_RESOURCE` constant in the class. This will map to one of the `id` attributes you configured in your module's `etc/acl.xml` (see 5.3). If you wish for more control over whether or not this controller is allowed, override the controller's `_isAllowed()` method.

1.06 Describe Magento CLI commands

Magento Objective: 2.02 Describe Magento CLI commands

Describe the usage of bin/magento commands in the development cycle. How are commands used in the development cycle?

`bin/magento` represents an easy way to interact with Magento. The CLI is trusted as it is expected that only trustworthy individuals have access to the CLI (as said person would also

have access to all code).

Here are the most common commands that I use (click on command for DevDocs):

- `bin/magento setup:install` : instead of trying to install Magento through the `/setup` url, use this. You will need to specify several basic options:

```
bin/magento setup:install \
  --base-url=https://lc.associate.site/ \
  --db-host=localhost \
  --db-name=associate \
  --db-user=root \
  --db-password=db_password \
  --admin-firstname=Joseph \
  --admin-lastname=Maxwell \
  --admin-email=joseph@swiftotter.com \
  --admin-user=admin \
  --admin-password=very-secret-password
```

- `bin/magento admin:user:create` : if you lose your admin password or are switching between systems and are unable to login, just reset your password.

- `bin/magento module:status` and `bin/magento module:enable` : when you create a module, you must enable it. Otherwise, you will be wondering why your changes aren't taking effect. To see if your module is enabled, check its status.

- `bin/magento setup:upgrade` : this runs setup install scripts and synchronizes the DB Schema. If you run this regularly, you can speed up your development environment by appending the `--keep-generated` flag.

- `bin/magento setup:static-content:deploy` : this builds the contents of the `pub/static` directories so your website becomes browseable. Note that this normally

is only run in production mode. By default, Developer mode does not allow this to command to be executed. Instead, use `bin/magento dev:source-theme:deploy` to utilize symlinks from the module's source files to their location in the `pub/static` directory.

- `bin/magento dev:source-theme:deploy`: this establishes symlinks for the LESS files. Symlinks allow your LESS files to always be kept up to date. Note that this command is also used to rebuild the `.css` files for a given theme.

- `bin/magento cache:flush`: no guide to Magento would be complete without a nod on how to flush the cache. I try to develop with as many caches enabled as possible. That isn't always possible. When I am building out a theme, I disable the full-page cache, block, and possibly layout XML. If I am not making regular changes to cached elements, leaving them enabled ensures fast(er) response times.

- `bin/magento indexer:reindex`: this reindexes all or the specific indices requested.

  You can also append a specific index (or multiple, separated by spaces) to reindex. *How do I know what the intended index's name is?* Here's what I do: I enter: `bin/magento indexer:reindex asdf` (ie, an invalid index name). The result: a list of all available index names that you can refresh. This is easier than trying to remember the names.

  If you have an index configured to run on Update on Schedule but cron is not functioning, the index will not be updated. However, having indexes configured to run on Update on Save can bring a local machine to a standstill as this can require significant CPU resources.

Side note: you can shorten these commands down to a combination that only exists for that command. For example, `cache:flush` can be executed as `c:f` (as there are no other commands that start with `c` and a `f`), or as `ca:fl` or as `cach:flu`. If you use an

abbreviation that exists elsewhere, you will be greeted with a (un)welcoming error.

For example:

- `bin/magento set:up` to run `bin/magento setup:upgrade`
- `bin/magento c:f` to run `bin/magento cache:flush`
- `bin/magento set:sta:dep` to run `bin/magento setup:static-content:deploy`

Which commands are available?

Run `bin/magento`.

Note that if you are trying to run a common command (like clearing the cache), and Magento says this command is not available, there is likely a problem with running the console. In this case, run just `bin/magento` and the error will appear.

Further reading:

- CLI Common Arguments

1.07 Describe index functionality

Magento Objective: 2.07 Describe index functionality

The core idea of indexing in Magento 2 is improving performance of some read operations at the cost of data redundancy.

Let us look at prices as an example. Pricing logic in Magento is complicated and tricky. Magento uses many different types of prices and price modifiers, for example:

- Price
- Tier price
- Custom option's price
- Catalog price rule

- Special price

The situation gets even more difficult when other product types, like configurable or bundle products come into play.

Of course there are formulas to derive the final price, but it causes multi-stage calculations which are impossible to perform efficiently for many products simultaneously. It is important to mention that assembling the final price on the fly for a single product is not a problem at all. The problem arises when we need to fetch prices for the whole catalog at a time, which is the case for sort/filter by price function on a category page. Since this functionality is so important, we can not ignore it, or tolerate a low performance while sorting.

The solution is to calculate the single value `final_price` beforehand and store it somewhere, so that, when we need to sort, we take that value and use a usual MySQL sort which is capable of handling a large number of records.

The problem, though, is that `final_price` is a subject of frequent changes, and it is redundant in the sense that it is just a result of calculation which uses data from other places. So whenever the original, raw date changes, one has to update `final_price`.

This is the core idea of indexing, which is the calculation process itself. MySQL tables store calculated data called indexes (don't mess with MySQL native indexes!).

Magento has a couple of indexes; you can view them by executing this CLI command:
`$ php bin/magento indexer:info`

Here is an example of the output:

```
design_config_grid                  Design Config Grid
customer_grid                       Customer Grid
catalog_category_product            Category Products
catalog_product_category            Product Categories
catalogrule_rule                    Catalog Rule Product
catalog_product_attribute           Product EAV
cataloginventory_stock              Stock
inventory                           Inventory
catalogrule_product                 Catalog Product Rule
catalog_product_price               Product Price
catalogsearch_fulltext              Catalog Search
```

Magento has a complicated indexing engine which defines when and how an index has to be updated. Technically, a developer can use it to create new indexes, but usually it is not needed.

The exact indexing mechanics is beyond the scope of this guide (and exam), but we will cover some general aspects of the process.

First, Magento tracks the changes in data to be indexed using special MySQL tables called "change_log"; you can find them by the suffix `_cl`. Usually, Magento uses MySQL triggers for that purpose, so a change log works on the MySQL level, when a piece of data is recorded into the database. Later, usually by a cron job, Magento runs the indexing process itself, which is a series of MySQL queries that transform data from change logs into the final indexed values. It is possible, though, to configure Magento to re-index an entity immediately after the change.

Further reading:

- Indexing Overview
- Index Management

1.08 Describe localization

Magento Objective: 2.08 Describe localization

Magento 2 is using two translation mechanisms: inline translations and dictionaries. The first one is a rather limited fancy tool that allows translating given phrases directly on the website. The second is a comprehensive framework that allows translating of everything: phtml templates, phrases hardcoded in PHP files, JavaScript, html templates, phrases in xml files (like layout or various configuration files).

The core idea is that all the phrases to be translated can be crawled by a special script which then generates a csv file with all the phrases to translate. The file only contains phrases, and it is the job of the translator (a human) to create translated pairs and write them in the csv file. Then, you upload the translated csv file back to the system, and Magento automatically substitutes translated phrases instead of original ones.

In order to find the strings to translate, the crawler needs some hints. The two principal hints are the “translate” attribute in xml files and use of the `__()` function to wrap the strings to translate. For example:

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

PHTML:

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

Similar approach is used in email templates, but instead of `__()`, we use `{{ trans … }}`.

Email templates:

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

You might have noticed that translatable strings may have placeholders, like `<?= __("There are %s items in your cart", $count); ?>`.

This significantly simplifies the translation process, since it allows a translation of a holistic phrase rather than separate pieces, which could be difficult to understand without a context.

In terms of xml files translation, the following examples illustrate the idea:

etc/di.xml

```
<type name="...">
    <arguments>
        <argument name="options" xsi:type="array">
            <item name="option1" xsi:type="array">
                <item name="value" xsi:type="string">container1</item>
                <item name="label" xsi:type="string" translatable="true">
                    Product Info Column
                </item>
            </item>
            <item name="option2" xsi:type="array">
                <item name="value" xsi:type="string">container2</item>
                <item name="label" xsi:type="string" translatable="true">
                    Block after Info Column
                </item>
            </item>
        </argument>
    </arguments>
</type>
```

etc/adminhtml/system.xml

```
<section
        id="catalog"
        translate="label"
        type="text"
        sortOrder="40"
        showInDefault="1"
        showInWebsite="1"
        showInStore="1">
    <class>separator-top</class>
    <label>Catalog</label>
    <tab>catalog</tab>
    <resource>Magento_Catalog::config_catalog</resource>
</section>
```

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

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

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

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

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

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

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

When run with the `-m` flag, two additional columns are added: `type` and `module`. `type` is

either `theme` or `module`. The `module` column represents the module that utilizes this translation.

Theme translation dictionary

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

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

Further reading:

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

Language packs

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

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

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

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

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

includes a `language.xml` file:

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

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

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

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

Further reading:

- DevDocs: Create a Language Pack
- German Locale Language Pack
- `i18n/Magento/fr_FR`

Database translations

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

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

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

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

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

Translations are applied in this order:

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

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

Further reading:

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

1.09 Describe Cron Functionality

Magento Objective: 2.03 Describe Cron Functionality

Cron functionality allows running some scripts in the background, outside of a regular client-side HTTP communication. If you are new to cron you can read this article. Cron is a system feature of all UNIX-based OS (including MacOS).

Magento has its own cron mechanism, which is synchronized with the system cron. There is a file `pub/cron.php`, which is supposed to be triggered by a system cron. The frequency is determined by a system administrator, and it's usually once a minute, once per 5 minutes or once per 15 minutes.

Once `cron.php` is executed, Magento pulls a list of scheduled (in Magento) cron jobs to be executed at a given time and executes them. Cron job is technically a PHP class, usually located in the Cron folder of a module. Note, that cron jobs have to be properly configured in order to be executed with the Magento cron jobs system (see below).

Another important note, is that all cron jobs are executed in the crontab area by default, which may cause some problems, especially related to the observer registered in the frontend or adminhtml area.

How are scheduled jobs configured?

Scheduled jobs (cron) is like an event, but it runs as often as every minute. You can configure the interval of how often your code is to be triggered. Please note that your cron job will run no more often than the Magento cron file is run (if your cron job is to be run every minute and Magento's `bin/magento cron:run` is triggered by the system cron every five minutes, your job will be run every five minutes).

You can also configure how often each cron group runs by going to Store Configuration > Advanced > System.

Add a `app/code/Vendor/Module/etc/crontab.xml` file to your module:

```
<?xml version="1.0"?>
<config xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:noNamespaceSchemaLocation="urn:magento:module:Magento_Cron:etc/crontab.xsd">
    <group id="default">
        <job name="export_order_data"
                instance="SwiftOtter\OrderExport\Cron\ExportOrders"
                method="execute">
            <schedule>* * * * *</schedule>
        </job>
    </group>
</config>
```

The schedule will execute no more often than your cron is configured on the hosting environment. For example, if you utilize the above example, and your system's cron is set to `*/5 * * * *`, the above will only execute every 5 minutes. This is not ideal, as you might guess.

Suggestion: to make testing custom cron jobs easier, I always create a console command and call my cron job from the console command.

For example: Create `app/code/SwiftOtter/OrderExport/Console/Command/CronRun.php`. Inject `SwiftOtter\OrderExport\Cron\ExportOrders` into `CronRun`'s constructor. In the console command's `execute` method, call `$this->exportOrders->execute()`. You can configure PHPStorm to run a PHP Script, which is an easy way to debug this action.

Further reading:

- DevDocs - Configure and Run Cron
- DevDocs - Custom Cron Jobs & Groups
- Amasty - Cron Job Starter Guide

- Cronitor - Crontab Guru

1.10 Describe custom module routes

Magento Objective: 2.10 Describe custom module routes

Native Magento routing system consists of two parts—Standard routing and Url Rewrites. Url Rewrites are covered in 2.11. Standard routing provides a framework to handle standard three-chunk urls, like `catalog/product/view/`. In order to process such requests, a module has to declare a route name (catalog) in the module's `etc/frontend/routes.xml` file declaring a new route and its front name as in the catalog's example:

```
<router id="standard">
            <route id="catalog" frontName="catalog">
                <module name="Magento_Catalog" />
            </route>
</router>
```

After that one creates a controller that corresponds to the `product/view` part of the `catalog/product/view` url. See 2.05 for more details about controllers.

It is possible, however, to introduce a completely custom url schema. For example, maybe instead of `/catalog/product/view/id/123` our system uses something like `product-123` or `product-configuration-ML-12-size-XL-color-RED`.

This requires creating a new Router. For doing so, it is important to understand what the Magento Front Controller is and how Magento routing works behind the scenes. In a nutshell, when a request hits Magento's `index.php`, it starts the application initialization process and then runs the Front Controller class. This class obtains a list of all registered (via `di.xml`) routers and loops through it to find the router that will respond to a given request. See: `\Magento\Framework\App\FrontController`

Normally Standard, CMS, or UrlRewrite routers respond to most of Magento native requests. However, it is possible to create a custom Router that will handle a custom URL schema. So, in order to create a custom URL schema, you should:

- 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`.

Further reading:

- DevDocs: Routing
- See section 2.05 for more details about controllers and URLs in Magento.

1.11 Describe URL rewrites

Magento Objective: 2.11 Describe URL rewrites

How is the user-friendly URL of a product or category defined? How can you change it?

The user-friendly URL for a product, category, or CMS page is found in the URL key attribute or column. If no URL key is specified, Magento will try to create the URL key by slugifying the name.

Magento will then create the applicable rows and add them to the `url_rewrite` table. For products, if the "Use Categories Path for Product Urls" is selected, Magento will also create URL paths that include the applicable category URL keys, separated by a slash (`/`).

How is a URL processed?

Before a controller is executed, there are a number of steps. The interesting thing is that a controller is executed by a router, and you can create your own router. We won't dig too

deeply into this, but here is an overview.

`pub/index.php` calls the `Bootstrap` per `\Magento\Framework\App\Bootstrap` which calls the `launch()` method on the current application class (`\Magento\Framework\App\Http` for a web request). This class uses the `ObjectManager` to initialize the `\Magento\Framework\App\FrontControllerInterface`, which returns a `FrontController`.

The front controller works like this:

- While the request has not been marked as being dispatched, the controller iterates through each router and checks if that router will match the request.
- When a match happens, the `foreach` is broken, and the entire list of routers will be rechecked.

If you are creating a custom router (Custom Routers), it is important to call `setDispatched()` on the request object to stop the first point's `while` loop.

Practical experience:

- Set a breakpoint in `\Magento\Framework\App\FrontController::dispatch()`.
- Navigate to a page on your local development environment.

That is the long story of how a URL is processed. However, all of that is necessary to get to this point. Hopefully you have taken the time to follow through with the practical experience. When you do that, you will see that these routers are executed in the following order (for a vanilla Magento installation):

- Robots.txt (`\Magento\Robots\Controller\Router`)
- URL Rewrites (`\Magento\UrlRewrite\Controller\Router`)
- Standard, what locates and executes a controller (`\Magento\Framework\App\Router\Base`)

- CMS, what watches for a CMS page (`\Magento\Cms\Controller\Router`)
- Default router, in other words, a 404 (`\Magento\Framework\App\Router\DefaultRouter`)

The URL Rewrite module's `di.xml` file is a great example of how to configure the sort order of a router. Notice that this router declaration is in the `frontend` area.

How do you determine which page corresponds to a given user-friendly URL?

The easiest way is to browse to the `url_rewrite` table and search for one of the URL segments in the `request_path` column. You can then find the product's ID in either the `entity_id` column (make sure the `entity_type` matches what you were anticipating) or the `target_path` column.

You can also use the `url_rewrite` table to not only map or rewrite paths, but also to redirect with a permanent (301) or temporary (302) redirect. However, visitors will appreciate faster redirects if you use Nginx or Apache configuration.

1.12 Describe the Magento caching system

Magento Objective: 2.12 Describe the Magento caching system

Magento caching system is crucial for its (Magento) proper functioning. The most important elements that are cached by Magento are:

- Various configuration xml files (`layout` , `configuration` , `ui_components` and so on)
- Some html output (blocks output, some of the pages can be cached completely)
- Information that is critical for internal operations like—schema info (tables, column names and types), information about attributes, entities, and so on

The Magento caching system is based on the `Zend_Cache` component. It consists of two

important pieces: cache frontend and cache backend.

- `\Magento\Framework\Cache\Frontend\Adapter\Zend`
- `Magento/Framework/Cache/Backend/`

Frontend provides an interface for developers to work with, and backend defines how exactly cache is stored.

Further reading:

- Configure Caching

———————————————————————

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:

- Manage the Cache
- Cache Overview

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: vendor/magento/framework/DB/Adapter/Pdo/Mysql.php

`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 would you clean the cache?

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

Further reading:

- Manage 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 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.

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`.

1.13 Describe stores, websites, and store views (basic understanding)

Magento Objective: 2.13 Describe stores, websites, and store views (basic understanding)

One of the key notions in Magento is that of **scope**. The idea behind scope is that some variables may have different values in different circumstances. The most prominent examples are prices and translations. We may have a product that is available in different locations and has everything the same but a price. It can be a different price or the same price but in different currency. Anyway, the value is different.

The situation with translations is similar. For example, a product's color which is a string, i.e. “red”, should be rendered in different ways when using multiple localizations. We can say that a “scope phenomenon” in Magento 2 is an ability for attributes to have different values.

Primarily we restrict our attention to EAV attributes and system configuration options (localization of static phrases works in a different way, see 2.08).

Magento has three elements in its scopes structure: Websites, Stores Groups, Stores. Please note, there is a mess in terminology: Store Group and Store are terms from the source code and used typically by developers, while Store and Store View (instead of Store Group and Store) is what merchants see, so Store and Store Views are typical for business audiences.

These three elements Website > Store Group > Store are hierarchical, which means each store belongs to some store group which in turn belongs to some website. A website may have

many child store groups, and a store group may have many child stores (but not the other way around: this is a one-to-many relationship).

Now, the next tricky thing is that Store Groups do not participate in this scoping functionality directly! Actually, a Store Group may have its unique root category and play a role of a container for multiple stores.

Websites and stores are more functional in the sense that EAV attributes and configuration options may have different values for websites/stores. It is important to understand that each EAV attribute has its scope—the property specified in the `is_global` field of the `catalog_eav_attribute` table. `is_global` is not a flag; it accepts three values—0, 1, 2 which state for store, global, and website.

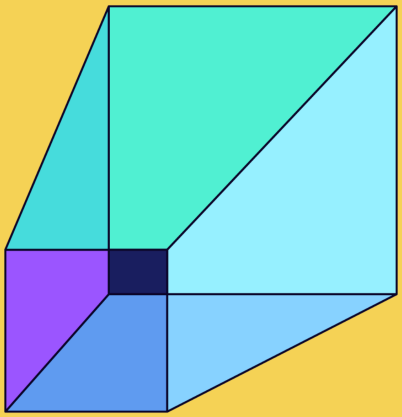
So, if an attribute has its `is_global` property set to 2, this means it has different values for different websites. If it is set to 1, then the attribute is global, which means no separated values are allowed, and scope 0 means that the attribute may have different values for different stores.

The values themselves are stored in corresponding EAV tables, like `catalog_product_entity_int`.

For system configuration options, all the values for all scopes are stored in `core_config_data` table.

Further reading:

- Websites, Stores, and Views



Working with Admin

Navigating basic Admin-area changes.

Study Guide for Professional Developer Certification, AD0-E711

SwiftOtter

2.01 Describe how the ACL works with roles and resources

Magento Objective: 1.01 Describe how the ACL works with roles and resources

Magento uses the Access Control List (ACL) to restrict permissions of its admin users as well as API users to perform certain actions. ACLs are specified for different endpoints in `etc/webapi.xml` files. For admin routes, ACLs are usually "embedded" into an admin controller using either the `ADMIN_RESOURCE` constant or the `_isAllowed` method.

```
class Save extends \Magento\Backend\App\Action
implements HttpPostActionInterface
{
    /**
     * Authorization level of a basic admin session
     * @see _isAllowed()
     */
    const ADMIN_RESOURCE = 'Magento_Cms::save';
```

See the whole file: `Magento\Cms\Controller\Adminhtml\Page\Save`

In order to define an ACL, we use an `etc/acl.xml` configuration file of a module.

How would you add a new ACL resource to a new entity?

If a module adds any functionality to the admin area, it is important to configure the Access Control List (ACL) such that an administrator can grant or deny access to this feature.

For example, as we create an order export tool, we would want to give the administrator the ability to determine who has access to exporting orders. Every module should have an `acl.xml` with at least one resource configured.

We configure this in our module's `etc/acl.xml`:

```
<?xml version="1.0" ?>
<config xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:noNamespaceSchemaLocation="urn:magento:framework:Acl/etc/acl.xsd">
    <acl>
        <resources>
            <resource id="Magento_Backend::admin">
                <resource id="SwiftOtter_OrderExport::export"
                        sortOrder="10"
                        title="Export Orders"
                />
            </resource>
        </resources>
    </acl>
</config>
```

Please note that you must place your ACL structure inside a `Magento_Backend::admin` resource. If you omit this, administrators will not see your ACL entry (and that pretty well defeats the entire purpose of utilizing ACL).

I am not aware of any requirements for ACL `id` naming. However, convention is to combine the module's name (as seen in `registration.php`) and the description of this action, separated by two colons `::`.

Further reading:

- Understanding ACL Rules

How do you manage the existing ACL hierarchy?

If you would like to prevent an existing ACL entry from appearing, you can disable it:

```
<resource id="Magento_Braintree::get_client_token"
        disabled="1" />
```

While this answers the question, I have never used this and can't think of a situation where this would be valid.

You can find the specification for this file in `Magento/Framework/Acl/etc/acl.xsd`.

2.02 Identify the components to use when creating or modifying the admin grid/form

Magento Objective: 1.02 Identify the components to use when creating or modifying the admin grid/form

Admin grids

Standard admin grid assumes there is an underlying table and collection that allows fetching data from the table. In order to create a grid itself one has to:

1. Create a new controller in `adminhtml`.
2. Modify layout to include the UiComponent's configuration.
3. Create a `listing` (grid) component xml file.
4. Create/configure the `DataSource` for a grid.
5. Create controllers for `massActions`.
6. Optionally - create a controller for inline editing.

Here is an example of a ***layout xml declaration*** (from the Magento_Cms module)

```
<referenceContainer name="content">
    <uiComponent name="cms_page_listing"/>
    ...
```

Example of the cms_page_listing.xml file:

`Magento/Cms/view/adminhtml/ui_component/cms_page_listing.xml`

Example of ***data provider*** configuration in di.xml:

```
<type name="Magento\Framework\View\Element\UiComponent\DataProvider\CollectionFactory"
    <arguments>
        <argument name="collections" xsi:type="array">
            <item name="cms_page_listing_data_source" xsi:type="string">
                Magento\Cms\Model\ResourceModel\Page\Grid\Collection
            </item>
            <item name="cms_block_listing_data_source" xsi:type="string">
                Magento\Cms\Model\ResourceModel\Block\Grid\Collection
            </item>
        </argument>
    </arguments>
</type>
<type name="Magento\Cms\Model\ResourceModel\Page\Grid\Collection">
    <arguments>
        <argument name="mainTable" xsi:type="string">cms_page</argument>
        <argument name="eventPrefix" xsi:type="string">
            cms_page_grid_collection
        </argument>
        <argument name="eventObject" xsi:type="string">
            page_grid_collection
        </argument>
        <argument name="resourceModel" xsi:type="string">
            Magento\Cms\Model\ResourceModel\Page
        </argument>
    </arguments>
</type>
```

See the whole file: `Magento/Cms/etc/di.xml`

And a ***special Grid Collection*** (note the implementation of `SearchResultInterface` - this is the main reason for having a separate collection. One may use the standard collection as well if implementing this interface): `Magento\Cms\Model\ResourceModel\Page\Grid\Collection`

Massactions are configured in the `listing/listingToolbar/massaction/action` nodes:

```
<massaction name="listing_massaction">
    <action name="delete">
        <settings>
            <confirm>
                <message translate="true">
                    Are you sure you want to delete selected items?
                </message>
                <title translate="true">Delete items</title>
            </confirm>
            <url path="cms/page/massDelete"/>
            <type>delete</type>
            <label translate="true">Delete</label>
        </settings>
    </action>
    <action name="disable">
        <settings>
            <url path="cms/page/massDisable"/>
            <type>disable</type>
            <label translate="true">Disable</label>
        </settings>
    </action>
    <action name="enable">
        <settings>
            <url path="cms/page/massEnable"/>
            <type>enable</type>
            <label translate="true">Enable</label>
        </settings>
    </action>
    <action name="edit">
        <settings>
            <callback>
                <target>editSelected</target>
                <provider>
                    cms_page_listing.cms_page_listing.cms_page_columns_editor
                </provider>
            </callback>
            <type>edit</type>
            <label translate="true">Edit</label>
```

Note urls to support massactions (see these examples: `Magento\Cms\Controller\Adminhtml\Page\MassDelete`, `Magento\Cms\Controller\Adminhtml\Page\MassEnable`), and that some massactions (edit, for example) may trigger a js module rather than issuing an immediate request.

For enabling an ***online editor***, one has to configure the `editorConfig` node in the `listing` xml file, configure the editor for each field and create a controller to save the result.

See an example and more details of configuring the editor here: ColumnsEditor Component

Pay attention to the `columns/settings/editorConfig` node:

```
<columns name="columns">
    <settings>
        <editorConfig>
            <param name="indexField" xsi:type="string">entity_id</param>
            <param name="enabled" xsi:type="boolean">true</param>
            <param name="selectProvider" xsi:type="string">
                ${ $.columnsProvider }.ids
            </param>
        </editorConfig>
    </settings>
    ...
```

And editor for each field:

```
<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>
```

See `Magento\Cms\Controller\Adminhtml\Page\InlineEdit` as an example of a controller associated with inline editor.

Admin forms

Forms follow the similar pattern to a grid:

1. Create a set of new controllers in adminhtml for the form itself, edit, save and delete actions. You may need additional controllers for files uploading, should there be ones.
2. Modify layout to include UiComponent's configuration.
3. Create an instance of a form ui component xml file.
4. Create/configure DataSource for a form.

See `Magento/Cms/Controller/Adminhtml/Page/` for examples of ***controllers***.

Layout modification is exactly the same as that of grids.

Form's ui component xml generally follows the same concept as listing, see here for more details: Form Component

DataSource configuration for a form is shown on the code snippet (of di.xml) above.

2.03 Identify the files to use when creating a store/admin config and menu items

Magento Objective: 1.03 Identify the files to use when creating a store/admin config and menu items

How would you add a new system configuration option?

System configuration options are configured in `etc/adminhtml/system.xml` (see `Magento/Catalog/etc/adminhtml/system.xml` for an example).

Note: Since these are XML files, you must clear the cache for Magento to recognize changes to these files. Bear in mind that when you do clear the cache, you will experience a longer-than-normal load time. This can be a problem in production.

```
<config xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:noNamespaceSchemaLocation="urn:magento:module:Magento_Config:etc/system_file.x
    <system>
    </system>
</config>
```

As you can see, the above doesn't do much. It's the basic requirement for a `system.xml` file, and we must build some content.

Create a tab:

```
<tab id="catalog" translate="label" sortOrder="200">
    <label>Catalog</label>
</tab>
```

Create Tab

Side note: this thought is just a good practice tip. If you work for a module vendor or an agency, please put all configuration choices in a pertinent area in Store Configuration. Do NOT put it under a tab for your company. For example, if you create an Order Attribute module, put configuration under Sales > Order Attributes and NOT `[MODULE VENDOR]` > Order Attributes. It makes administration much easier.

Create a section:

The `section` tag lives inside the `system` tag (not a `tab` element).

```
<section
        id="catalog"
        translate="label"
        type="text"
        sortOrder="40"
        showInDefault="1"
        showInWebsite="1"
        showInStore="1">
    <class>separator-top</class>
    <label>Catalog</label>
    <tab>catalog</tab> <!-- this matches the tab ID attribute -->
    <resource>Magento_Catalog::config_catalog</resource>
    <!-- this matches the etc/acl.xml's resource ID attribute -->
</section>
```

Create Section

The section appears under the tab headings.

Create a group:

This happily resides in the `section` tag.

```
<group
        id="recently_products"
        translate="label"
        type="text"
        sortOrder="350"
        showInDefault="1"
        showInWebsite="1"
        showInStore="0">
    <label>Recently Viewed/Compared Products</label>
</group>
```

You can also nest groups. This is seen on the Store > Configuration > Sales > Payment Methods > Authorize.net page (see example: `Magento/AuthorizenetAcceptjs/etc/adminhtml/system.xml`).

When nesting groups, you want to use the `config_path` value for a field to configure where this value is stored:

```
<config_path>payment/authorizenet_acceptjs/title</config_path>
```

Create an entry:

Finally! Let's add a configuration field with the `field` tag (found inside a `group` element):

```
<field
        id="synchronize_with_backend"
        translate="label"
        type="select"
        showInDefault="1"
        canRestore="1">
    <label>Synchronize widget products with backend storage</label>
    <source_model>Magento\Config\Model\Config\Source\Yesno</source_model>
</field>
```

Create Entry

This example uses the `showInDefault` attribute. This means the attribute is visible in the global scope. You can also use the `showInWebsite` or `showInStore`.

In this example, we have created a dropdown or "select" list. There are quite a few types of fields, which we will discuss in the next section.

To access these values:

- Inject an instance of `ScopeConfigInterface`.
- Call `$this->scopeConfig->getValue('...')`.
- The scope configuration path is realized by concatenating the section `id`, the group `id`, and the field `id`. For example, `catalog/recently_products/synchronize_with_backend` (as seen in the previous code excerpts).
- In the `getValue`, you can also load a value for a particular scope:
    - The second parameter is which type of scope (\Magento\Store\Model\ScopeInterface). You can use `SCOPE_WEBSITE` or `SCOPE_STORE` (which really is a store view).

- The third parameter is the identifier for the scope. If you need a configuration associated with a particular order, use the `$order->getStoreId()` method.

Note: a common mistake that I see is not to consider the scope and scope ID when loading values from store configuration. This is often not necessary on a frontend template. However, in the admin, this is often important unless the scope for a configuration setting is global.

What is the difference in this process for different option types (secret, file)?

There are several values for the `type` attribute. This attribute determines what type of input is used for a given option.

- `type="text"` : shows a single line text field. Ideally, this is used for values that a store administrator would want to control. This field type also will usually be visible in global, website, and store view scopes so that it can be translated.
- `type="select"` : shows a drop-down list. The values are specified in a `source_model` element nested inside the `field` element. The source model should implement `\Magento\Framework\Option\ArrayInterface`.
- `type="multiselect"` : shows a multiple-select list. This allows the user to select multiple values (countries, for example).
- `type="allowspecific"` : this powers the "Ship to Applicable Countries." This controls whether or not the Countries select is enabled.
    - `\Magento\Config\Block\System\Config\Form\Field\Select\Allowspecific`
- `type="obscure"` : presents a password input. You will also want to include the `backend_model` element in this `field` :

```
<backend_model>Magento\Config\Model\Config\Backend\Encrypted</backend_model>
```

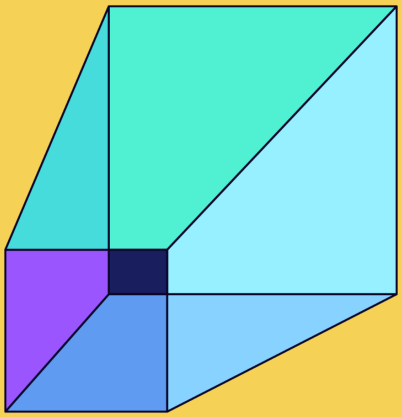
    - Note that there is a `password` type, but the value is not encrypted (see What is the Difference Between Password and Obscure Type).

There are also several configurations for fields. While you may not use them every day, these will come in handy from time to time:

- `config_path` (element): this allows you to map a field to a different configuration path. For example, if you need to move a setting in the configuration, but you don't want to update the path whereby the configuration value is referenced, you can use this option. Move the element but put the old path in the `config_path` element.
- `label` (element): the name of this field.
- `sortOrder` (attribute on `field`): configures the order in which these elements appear.
- `depends/field id="..."` (element): this makes the current field dependent.
- `validate` (element): apply validation on the element. To make a field required, use the `required-entry` value. You can add multiple validation classes separated by space.
- `backend_model` (element): specifies a class that inherits `\Magento\Framework\App\Config\Value`. This class configure functionality for before/after save and after delete.
- `frontend_model` (element): the block that renders the element. If you need to create a custom element, you would probably extend a class in the `\Magento\Config\Block\System\Config\Form` namespace.

Further reading:

- MagePlaza: System Configuration Field Types
- DevDocs: system.xml Reference



EAV/Database

Understanding the database and its structure.

Study Guide for Professional Developer Certification, AD0-E711

SwiftOtter

3.01 Given a scenario, change/add/remove attribute sets and/or attributes

See 3.02 and 3.05 for more details about different types of attributes and EAV attributes in particular.

EAV attributes in Magento are grouped into attribute sets. Technically, each instance of an EAV entity is connected to some attribute set. For example, each product has an attribute set. This is also true for categories and customers, but in practice they always stay within the single attribute set. So, attribute sets usually only differ for different products.

All attribute sets are stored in the `eav_attribute_set` table.

```
mysql> SELECT * FROM eav_attribute_set;
+------------------+----------------+--------------------+------------+
| attribute_set_id | entity_type_id | attribute_set_name | sort_order |
+------------------+----------------+--------------------+------------+
|                1 |              1 | Default            |          2 |
|                2 |              2 | Default            |          2 |
|                3 |              3 | Default            |          1 |
|                4 |              4 | Default            |          1 |
|                5 |              5 | Default            |          1 |
|                6 |              6 | Default            |          1 |
|                7 |              7 | Default            |          1 |
|                8 |              8 | Default            |          1 |
|                9 |              4 | Top                |          0 |
|               10 |              4 | Bottom             |          0 |
|               11 |              4 | Gear               |          0 |
|               12 |              4 | Sprite Stasis Ball |          0 |
|               13 |              4 | Sprite Yoga Strap  |          0 |
|               14 |              4 | Downloadable       |          0 |
|               15 |              4 | Bag                |          0 |
```

You can see that only the product entity (`entity_type_id=4`) has multiple attribute sets.

In addition to attribute sets, Magento is using attribute groups for grouping the attributes. Usually the attribute group is used on the form page to form accordions (Like Product Details, Images, Content and so on). Groups are stored in the `eav_attribute_group` table, and each group is connected to one parent attribute set. You can check the table and verify that all of the non-trivial groups are related to the product and category entities. Unlike attribute sets, groups do not have any important function other than rendering attributes on an edit page.

An association between attributes, entities, and groups is stored in the `eav_entity_attribute` table.

```
> SELECT * FROM eav_entity_attribute WHERE entity_type_id=4 LIMIT 10;
+---------------------+----------------+------------------+--------------------+-------
| entity_attribute_id | entity_type_id | attribute_set_id | attribute_group_id | attrib
+---------------------+----------------+------------------+--------------------+-------
|                  73 |              4 |                4 |                  7 |
|                  74 |              4 |                4 |                  7 |
|                  75 |              4 |                4 |                 13 |
|                  76 |              4 |                4 |                 13 |
|                  77 |              4 |                4 |                  7 |
|                  78 |              4 |                4 |                  8 |
|                  79 |              4 |                4 |                  8 |
|                  80 |              4 |                4 |                  8 |
|                  81 |              4 |                4 |                  8 |
|                  82 |              4 |                4 |                  7 |
+---------------------+----------------+------------------+--------------------+-------
```

You may notice that this table is denormalized, since `attribute_group_id` carries information about `attribute_set_id` and `entity_type_id` .

Now, attribute sets management can be done in one of two ways: in the admin panel and programmatically. Regarding admin panel read: Attribute Sets.

If you want to add an attribute set programmatically, there is one subtle detail to be aware of. Normal attribute sets (for product) must include some attributes (like price) for a product to be functional. This means that creating a new, empty attribute set is useless, unless we assign these attributes to it. Fortunately there is a possibility to create an attribute "based" on another attribute set, which means attributes from the parent set will be copied to its children. This is done by the `initFromSkeleton()` method of the attribute set's model. For example: How to Create an Attribute Set Programmatically & Assign Attributes to It.

See 3.05 for information about EAV attributes management.

3.02 Describe different types of attributes

Attributes are an important feature of Magento. Many out-of-the box features rely on attributes heavily, and it is possible to create new attributes to implement custom functionality. What makes attributes so powerful in Magento, is that they go far beyond mere elements of data architecture. Attributes in Magento are functional. This means you can use attributes to change a behavior of some object, rather than simply adding another property to it.

There are two primary types of attributes: EAV attributes and extension attributes. The next table summarizes some of their properties:

	EAV attribute	Extension attribute
Applied to	A Resource model that extends `Magento\Eav\Model\Entity\AbstractEntity`	Data model that extends `Magento\Framework\Api\AbstractExtensibleObject`, or model that extends `Magento\Framework\Model\AbstractExtensibleModel`
Values stored	In special tables with the types `*_entity_varchar`, `*_entity_int` and so on. For example:	It is up to developer to decide where to store the data

	EAV attribute	Extension attribute
Save and load	`catalog_product_entity_int` Automatically implemented in the `AbstractEntity` resource model. A lot of functionality is implemented in `Magento_Catalog` abstract resource model	Manually. Developer has to load and save the data
Use for a custom entity	Very difficult	Easy
Create new attribute	Requires a DataPatch	Requires an entry in `etc/extension_attributes.xml` file
Availability in WebAPI	Available as Custom Attributes	Available as Extension Attributes
Scope support	Out of the box for product and category attributes	Does not support scope out of the box, up to developer to implement
Typical use case	Extend an entity with a new scalar property, implement some functionality when the property changes, support different values for different scopes	Extending an entity with a new property, not necessarily scalar, could be an object or array. Data is stored in custom tables, files, could even be fetched by API
Grid and form support	Yes, out-of-the box	No, requires custom work

Further reading:

- Adding Extension Attributes to Entity

3.03 Given a scenario, use a DB schema to alter a database table

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. DB Schema is available as of Magento 2.3.

This configuration is found in your module's `etc/db_schema.xml` configuration file.

Example:

```
<!-- app/code/SwiftOtter/TestModule/etc/db_schema.xml -->
<schema xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
        xsi:noNamespaceSchemaLocation="urn:magento:framework:Setup/Declaration/Schema/
    <table name="price_list">
        <column xsi:type="int"name="id" unsigned="true"
                nullable="false" identity="true" comment="ID"/>
        <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`. This is where you can see all the available options.

Table: this identifies the table that is created (or modified). The `name` attribute is required. This determines the name of the table in the database. This is the element that is used for merging table configuration across multiple `db_schema.xml` files.

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.

How do you add a column using declarative schema?

As such, you can add a column into an existing Magento table (this adds a `delivery_date` column into the Magento `quote` table):

```
<?xml version="1.0"?>
<!-- app/code/SwiftOtter/OrderExport/etc/db_schema.xml -->
<schema xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
        xsi:noNamespaceSchemaLocation="...">
    <table name="quote">
        <column xsi:type="datetime" name="delivery_date" nullable="false"
                comment="Delivery Date"/>
    </table>
</schema>
```

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

- `name`: of course, the name of the column
- `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 column to the new column (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.

Practical experience: follow along in the example code to understand how to create tables/columns in the database.

How do you modify a table added by another module?

You would most likely be adding a column, as removing or changing the column's name will have possible serious consequences. In the section above, you can see how to create a column. In addition, you must ensure that you have configured the module load order (Component Load Order) in your module's `etc/module.xml` file. This will ensure that your module is executed after the module whose `db_schema.xml` file creates the original table.

That said, you can use the `disabled` attribute to remove a column, constraint, index or table from the database (or prevent it from being added).

How do you delete a column?

This code will remove the `coupon_code` column from the `quote` table.

```
<!-- app/code/SwiftOtter/TestModel/etc/db_schema.xml -->
<schema xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
        xsi:noNamespaceSchemaLocation="...">
    <table name="quote">
        <column name="coupon_code" disabled="true" />
    </table>
</schema>
```

How do you add an index or foreign key using declarative schema?

Use the `constraint` tag. There are two types of constraints in schema: indexes and foreign.

Indexes utilize the `constraint` tag but have a `column` class to build the structure of the index. 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:

```
<constraint
    xsi:type="foreign"
    referenceId="ORDER_EXPORT_DETAILS_ORDER_ID"
    table="order_export_details"
    column="order_id"
    referenceTable="sales_order"
    referenceColumn="entity_id"
    onUpdate="CASCADE"
    onDelete="CASCADE"
/>
```

Further reading:

- Configure Declarative Schema

How do you manipulate data using data patches?

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

You can initialize a patch with the CLI command:

```
bin/magento setup:db-declaration:generate-patch SwiftOtter_Test
```

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.

There are three 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 patch able 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 relying on patches and the more intuitive DB Schema to get the job done.

Further reading:

- Develop Data and Schema Patches

What is the purpose of schema patches?

Schema patches allow for intricate updates to the schema. I find that almost every situation is covered in `db_schema.xml`.

However, let's say you need to add a column to a table that is outside of Magento's control (like a custom-built application that, for some reason, shares the same Magento database).

The process for creating a schema patch is very similar with two exceptions:

- The patch should reside in your module's `Setup/Patch/Schema` directory.
- The patch should implement the

`\Magento\Framework\Setup\Patch\SchemaPatchInterface`.

You might be surprised that this interface has no methods. Why would an interface extend another if it doesn't add to the original interface's functionality?

Labeling. Because you would choose to implement the `DataPatchInterface` or the `SchemaPatchInterface`, Magento knows what type of patch this is *without having to rely on a directory structure*.

3.04 Describe models, resource models, and collections

There are four types of classes related to loading and storing data in Magento.

- Models: stored directly in the module's `Model/` directory: these are classes with getters and setters to store data. An instantiated class would represent one row in the database. This model is initialized with a reference to the Resource Model.
    - Example: `\Magento\Cms\Model\Page`
- Resource models: stored in the module's `Model/ResourceModel/` directory, these classes are responsible for saving and loading data. These classes almost always inherit `\Magento\Framework\Model\ResourceModel\Db\AbstractDb`. The primary methods in this class are aptly named: `load()`, `save()` and `delete()`. Please see the examples in the sample project for how to configure this class. However, the resource model is the place to put any custom selects (using the Magento ORM, and NOT writing direct SQL).

  Resource models are initialized with the table name (as found in `db_schema.xml`) and the name of the primary key column.
    - Example: `\Magento\Cms\Model\ResourceModel\Page`

- Collections: stored in the module's `Model/ResourceModel/[Model Name]/Collection.php` file, this class loads multiple models. This class is initialized with a reference to the Model and Resource Model.

  Collections store state. If you wish to utilize a CMS Page collection in one of your classes, you must use a Factory (`Magento\Cms\Model\ResourceModel\Page\CollectionFactory`) and instantiate that collection (`$this->collectionFactory->create()`) instead of injecting the collection class itself. See Objective 1 for more details about how to use factories.

    - Example: `\Magento\Cms\Model\ResourceModel\Page\Collection`

- Repository: this handles the primary actions that happen to a database table. Repositories usually have `save()`, `getById()` (which gets one row/model from the database), `getList()` (which accepts a `SearchCriteria` class), and `delete()`. Repositories do not inherit another class. They represent more of an idea (unlike models, resource models, or collections). There will be several classes injected, but the most common are the model's factory and a resource model. To create a repository, the easiest is to copy the CMS Block or CMS Page repositories. Or, see the example in our `ExportDetails` repository for a simplified version.

  Repositories are often a wrapper for the Resource Model. The latter has no way to instantiate a model: the repository creates the model (via the model's factory) and then passes it to the Resource Model's `load()` method.

  Repositories do not store state. They are essentially a wrapper to ease the effort of data operations.

  If you modify a repository, make sure that the output stays the same. For example, if you want to utilize a different model instead of a `\Magento\Catalog\Model\Product`, make sure that this new model implements the appropriate interface (in this case

`\Magento\Catalog\Api\Data\ProductInterface`). This is because there are many areas of Magento that rely on the product repository returning a `ProductInterface`. This is just for the sake of example as I expect you would spend many hours pulling out your hair while trying to make this change as there are too many dependencies on the implementation of the product interface.

Repositories are commonly an API Endpoint (the `service class` in a `webapi.xml` file). For this situation, you would create an interface that the repository then implements. You would declare the route and resources required.

How does a Repository's `getList()` method differ from a collection? For a normal model (like a CMS page or an order or order item), there would be no difference. However, for an EAV entity (like a product, category, or customer), there is a **big** difference: all attributes are loaded with a repository, but only the attributes you select with a collection. If you are loading many products, this can be a major performance slow down. Again, for EAV-enabled entities, you can select which attributes you want to be loaded. Most of the time, when you are iterating through a list of products, you will only use a few attributes.

So why is there this option? Originally, a repository was designed for use through an API: and I see APIs as less concerned about speed and more about how much information can be returned.

    - Example: `\Magento\Cms\Model\BlockRepository`

What are the responsibilities of each of the ORM object types?

- Data models store data. Once they are loaded from the database (hydrated), they are ready for use. Models that extend `\Magento\Framework\Model\AbstractModel` will thus extend `\Magento\Framework\DataObject`. This means that the data is stored in the class' `$_data` property.

You can convert snake-case notation (`discount_amount` as an example of a table's column name OR the attribute code) to camel-case notation (`DiscountAmount`) and get the discount amount like: `$class->getDiscountAmount()` . This is called a "magic getter". You can use a "magic setter" by calling `$class->setDiscountAmount($productId)` .

Or, you can use `$class->getData('discount_amount')` to likewise get the data. I generally do not use magic getters and setters. And, calling `getData` all over the place is somewhat clunky.

Instead, I regularly take the time to build out getters and setters.

```
public function getDiscountAmount(): float
{
   return (float)$this->getData('discount_amount');
}

public function setDiscountAmount(int $discountAmount): void
{
    $this->setData('discount_amount', $discountAmount);
}
```

The advantages of this are several: 1) you get type hints and this can reduce unnecessary conversions (`float` to `int` , for example). 2) you can mock these methods in your unit tests 3) you can customize how data is handled if you want to `json_encode` a value in a setter.

- Resource models handle database operations. They save and load by default. This functionality happens in the abstract class `Magento\Framework\Model\ResourceModel\Db\AbstractDb` .

I find that I can improve performance by writing custom queries in methods in these classes. For example, if I want to find the number of rows in a table that match a given criteria:

```
public function getAvailableToTranslate(string $language): int
{
    $select = $this->getConnection()->select();
    $select->from($this->getMainTable(), 'COUNT(id)');
        // leaving the second parameter blank is the equivalent of `*`.
    $select->where('needs_translation = ?', true);
    $select->where('language = ?', $language);

    return (int)$this->getConnection()->fetchOne($select);
}
```

This would be an example of getting the number of items that need to be translated in a fictitious module. Note that I use `fetchOne()` (`fetchOne()` method in sql query). This returns one result. You can also use:

- `fetchAll()` (`fetchAll()` method in sql query)
- `fetchCol()` (`fetchCol()` method in sql query)
- `fetchPairs()` (`fetchPairs()` method in sql query)
- `fetchRow()` (`fetchRow()` method in sql query)
- `fetchAssoc()` (`fetchAssoc()` method in sql query)

  This is often much more efficient than loading in rows with a collection and iterating/tallying up the results.

  You can also use resource models to create custom handling for updating or

inserting new rows. For example, if you are saving a list of rows that don't have primary keys (think saving product options), you can create a method to accept this array, iterate through each, and save. Again, this is a more efficient approach (considering server performance) than creating models for each row and calling save on each.

- Collections handle loading multiple rows for a database entity or table. As discussed previously, the lines begin to blur as to whether to use a repository or collection.

  Here are a few methods that are commonly used in collections (`\Magento\Framework\Model\ResourceModel\Db\Collection\AbstractCollection`):

  - `addFieldToFilter` : this is the equivalent of adding a `where` to a SQL select. See addAttributeToFilter Conditionals In Magento for more conditions.

    Example:
    `$collection->addFieldToFilter('product_id, ['neq', $productId);`

    ```
    SELECT *
    FROM table_name
    WHERE product_id <> 4;
    ```

    - `addAttributeToFilter()` (on EAV entities `Magento/Eav/Model/Entity/Collection/AbstractCollection.php`): this is the same as `addFieldToFilter()` for EAV-enabled entities. If you call either method, you get the same result. This has the same behavior as above.

  - `join()` : joins in another table.

    Example:

```
$collection->join(
  [‘table_alias’ => ‘table_name’],
  ‘main_table.product_id = table_alias.product_id’,
  [‘column_a’]
);
```

This results in:

```
SELECT *, table_alias.column_a
FROM original_table AS main_table
INNER JOIN table_name AS table_alias
   ON main_entity.product_id = table_alias.product_id;
```

Please note that `main_table` is the default alias with collections.

- `load()`: this fetches the results of the collection. Data is loaded from the database and then models are hydrated with said data.
- `getSize()`: this creates a copy of the `select`, strips out the columns, and uses the `COUNT` method. If you are making many customizations to collections, such as adding groups, you will likely have problems with this method and would want to use an `after` plugin to alter the results of this `select`.

Practical experience: I strongly recommend setting breakpoints in the resource model, collection, and model classes covered above. Go to the homepage of your demo site. Step through the breakpoints. This will educate you on what methods are used for what places.

How do they relate to one another?

A collection loads a list of models. Because collections implement the `\IteratorAggregate` interface, you can loop through a collection in a `foreach()` method. For practical experience,

I suggest you set a breakpoint in `\Magento\Framework\Data\Collection\AbstractDb::load()` and observe what methods trigger calling this.

A resource model directly interacts with the database. Ultimately, it handles loading (hydrating) a model class that is already instantiated. It will also save data back to the database.

A model stores data. `get()` and `set()` are used all day long.

3.05 Describe basics of Entity Attribute Value (EAV)

Magento EAV is a framework that allows entities to have different values for its properties. See 2.13 for discussion of scopes and multi-valued attributes. Another purpose of EAV is to provide a flexible mechanism to extend an entity's data architecture. For example, it allows laptops and mobile phones, which are instances of the `Catalog_Product` entity, to have different properties, which we call attributes.

EAV framework is very flexible, and allows easily attaching or detaching attributes from particular instances; see 3.02 for discussion of EAV attributes and how they relate to extension attributes.

EAV is technically implemented via ResourceModel (again, see 3.02 for details). Note, that it is very difficult to create a custom EAV entity, and usually there is no need for that.

EAV consists of the following components:

- Entity types
- Attribute sets
- Attributes
- Attribute values

Entity types.

Each EAV entity must be registered as an entity type in the `eav_entity_type` table.

```
SELECT entity_type_id, entity_type_code, entity_table FROM eav_entity_type;
+----------------+------------------+------------------------+
| entity_type_id | entity_type_code | entity_table           |
+----------------+------------------+------------------------+
|              1 | customer         | customer_entity        |
|              2 | customer_address | customer_address_entity |
|              3 | catalog_category | catalog_category_entity |
|              4 | catalog_product  | catalog_product_entity  |
|              5 | order            | sales_order            |
|              6 | invoice          | sales_invoice          |
|              7 | creditmemo       | sales_creditmemo       |
|              8 | shipment         | sales_shipment         |
+----------------+------------------+------------------------+
8 rows in set (0.00 sec)
```

You can see that there are eight entity types in Magento. They come from the early days of Magento 1, and by Magento 2.4, have evolved so that only `catalog_category` and `catalog_product` are fully-fledged entity types. There are still some features of EAV used for `customer` and `customer_address` entity types, but on a much smaller scale. Order and invoice entities, in turn, have "developed" some features, like increment model, that is technically a part of the EAV framework, but are not used by other entities.

Other entity types are rather rudimentary; there is pretty much nothing left from EAV in orders and invoices.

Another important aspect is that many EAV features are developed in the `Magento_Catalog` module (rather than in `Magento_Eav`). For example, celebrated multi-scope functionality of EAV attributes is a core feature of the EAV framework, however it is fully implemented in the Catalog module, which means, in particular, that Customer's attributes will have problems with multi-scoped values. Out of the box, all customer's attributes are global (and static). Run this

query to verify:

```
SELECT attribute_id,
       attribute_code,
       backend_type,
       backend_table
FROM eav_attribute
WHERE entity_type_id=1;
```

Attribute sets

See 3.01.

Attributes

Each `eav_attribute` has a lot of information associated with it, besides its values. That information can be generic or entity type specific. Generic information is stored in the `eav_attribute` table. Information specific for category and product attributes is stored in `catalog_eav_attribute`, and information specific for `customer` and `customer_address` is stored in `customer_eav_attribute`.

While it is possible to add attributes via an admin interface, we should try to do that programmatically via a Data Patch. Attributes added through the admin panel, will only be available in a single copy of the database, while those added by a Data Patch will be reproduced every time a new database is deployed.

Further reading:

- How to Add a New Product Attribute
- Adding Customer EAV Attribute for Backend Only

As we mentioned before, attributes have their own properties which define an attribute's

behavior. You see the list of the properties by checking `eav_attribute` or `catalog_eav_attribute` tables. Usually they are quite self-explanatory.

But we would like to discuss three of them, which are of great importance: backend, source, and frontend models.

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:

Product Attribute

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 you to specify options in 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 `\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 a 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.

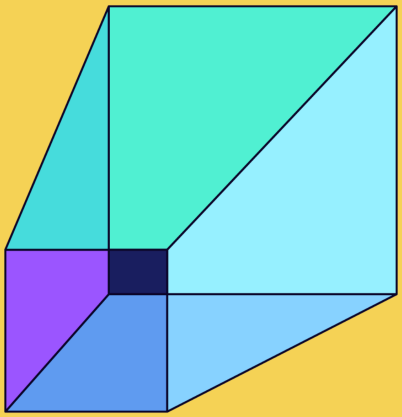
Attribute values

Finally we discuss EAV attribute values. Values are stored in a set of tables, specific per entity

type. For example for a catalog product:

```
> show tables like 'catalog_product_entity_%';
+-----------------------------------------------------+
| Tables_in_magento_224 (catalog_product_entity_%)    |
+-----------------------------------------------------+
| catalog_product_entity_datetime                     |
| catalog_product_entity_decimal                      |
| catalog_product_entity_gallery                      |
| catalog_product_entity_int                          |
| catalog_product_entity_media_gallery                |
| catalog_product_entity_media_gallery_value          |
| catalog_product_entity_media_gallery_value_to_entity  |
| catalog_product_entity_media_gallery_value_video    |
| catalog_product_entity_text                         |
| catalog_product_entity_tier_price                   |
| catalog_product_entity_varchar                      |
+-----------------------------------------------------+
11 rows in set (0.01 sec)
```

Each table stores `entity_id`, `attribute_id`, `value` and `store_id` which is an indicator for a scope. Note, that `store_id`'s interpretation depends on the attribute's scope: global, website, store (see 2.13 for scopes discussion). Say there is a record with `attribute_id=25`, `scope_id=4`, `value=133`. We have to check what is the scope of the attribute with id 25. For doing so we check `eav_attribute`, `is_global` field. Assume it equals 2, which means website. So the attribute has different values on different websites, and its value for a website with id 4 equals to 133.



Layout/UI

Knowing the basics of the presentation layer.

Study Guide for Professional Developer Certification, AD0-E711

SwiftOtter

4.01 Describe usage of CMS pages and blocks

CMS pages and blocks are two features that allow admin users to create static content without any development.

See DevDocs - Adding a New Page for more information about CMS page management.

When creating a CMS page and/or block the key is the content. It can be done with the use of PageBuilder or native CMS functionality.

Content pages, blocks, and widgets

Magento CMS capabilities consist of CMS pages, blocks, widgets, and PageBuilder.

CMS pages are standalone pages, usually with some static information, like “About Us”. They can be created in the Admin panel. CMS blocks are “pieces of html” that can be reused many times on different pages. Widgets are somewhat similar to blocks but more powerful and functional.

All elements are well-documented, and their admin management is more or less self-explanatory. Here we point out some Magento-specific features such as:

- Variables — Inserting a Variable
- Use of blocks developed on the backend (Template Blocks) — StackOverflow: Add Block to CMS Page
- Magento-specific commands, such as generating the URL taking into account store settings — Inserting a Link

Such Magento-specific features are usually added with `{{ … }}` in the text of the CMS page or block.

Further reading:

- CMS Pages

- CMS Blocks
- CMS Widgets

4.02 Given a scenario, modify layout

How do you use layout XML directives in your customizations?

Layout XML connects template files with a request. As discussed previously, we determine the layout handle through the request. There are some exceptions as a controller can create layout handles and add those to the request. A good example is seen by setting a breakpoint in `\Magento\Catalog\Helper\Product\View::initProductLayout()`. Navigate to a product page and step through this method.

Layout XML is found in each module's `view/[area]/layout` directory. The XML instructions are merged together to generate a massive XML tree. Files are read according to their module's order in `app/etc/config.php`. XML nodes can override or modify other nodes.

Here are the most common layout XML instructions that you will use:

- You can declare a block that will render HTML:

  ```
  <block name="order.export" template="SwiftOtter_OrderExport::form.phtml">
  ```

  To adjust the rendering order of a particular block, you can use the `before` and `after` attributes. A `-` value means the first or last, respectively. You can also use the name of another block in the current container to add before or after that block.

  There is often confusion about the difference between `as` and `name`. These attributes determine how to reference this block within the template. To access a child block in a template, you would use the value of the `as` attribute:

```
$block->getChildBlock('as.value');
```

If the value of the `as` attribute is not specified, use the value of the `name` attribute.

I find it much easier to never specify the `as` attribute as this introduces another level of complexity and can thus be confusing.

Remember, if you see the `as` attribute specified for a block, reference that block within a template by the value of the `as` attribute.

- You can reference an existing block.

```
<referenceBlock name="order.export">
    <!-- arguments/argument nodes here -->
</referenceBlock>
```

- A container renders all blocks and containers inside of it. You can wrap the contents in a desired HTML tag with the `htmlTag` attribute.

```
<container name="order.export.renderer" htmlTag="div" />
```

- You can reference an existing container to add your block to it:

```
<referenceContainer name="content">
    <block name="order.export" template=”SwiftOtter_OrderExport::form.phtml”>
</referenceContainer>
```

- You can `move` a block to a different parent:

```
<move element="order.export" destination="footer" />
```

- You can `remove` a block.

```
<remove name="order.export" />
```

- You can include another layout handle. This is done, for example, with the `customer_account` layout handle. When visiting pages within the customer account, like `vendor/magento/module-customer/view/frontend/layout/customer_address_form.xml`, you will see the `update` node:

```
<update handle="customer_account"/>
```

  This includes the layout as specified in all `customer_account.xml` files.

- You can prevent an entire page from being cached with the `cacheable="false"` attribute on a block. Never ever use this unless you are 100% confident that the entire page where this block is loaded does not need to be cached. I have seen plenty of cases where irresponsible module developers needlessly use it—at the cost of performance (and sales) on their merchant's website.

Further reading:

- Layout Instructions

How do you register a new layout file?

You first must determine the layout handle. Convert any dashes present to underscores, append a `.xml` file extension and ... presto! You have a new layout file.

The layout handle is determined in a couple of ways. The easiest is to look in the source and read the `<body/>` tag's `class` attribute.

This method doesn't always get every layout handle.

The best way to get all layout handles available in a request is to set a breakpoint in `\Magento\Framework\View\Result\Layout::addHandle()`. Visit your store's home page (you may have to flush the cache). Add a `.xml` to this layout handle and place the newly-created file in your modules `view/[area]/layout` directory.

As an example, loading a configurable product results in these layout handles:

- `default`
- `catalog_product_view`
- `catalog_product_view_type_configurable`
- `catalog_product_view_id_436`
- `catalog_product_view_sku_MJ12`

For any of these handles, you can create a layout XML file, like:

```
app/code/SwiftOtter/OrderExport/view/frontend/layout/
catalog_product_view_id_436.xml
```

Changes written in this file will only take effect on the product page with the ID 436.

4.03 Given a scenario, modify page style

Depending on a page and how it is implemented we can talk about multiple ways of modifying a page's style.

Page type	Required modification	Theme-level modification	Module-level modification
Core page	Change style defined in core less files	In a custom theme you can extend core styles (less files).	You can create a custom module with its own less file that modifies core styles.
Core page	Override the style defined in a core less file.	In a custom theme, you can override a less file from a parent theme.	Usually you use a custom theme when something has to be overridden.
Core page, Custom page	Add third-party css	Use layout update	Use layout update
Custom page	Add new less instructions	Use custom theme's less files	Use module's less file

As we see from the table we may need to:

1. Customize less file defined in a parent theme/core module using a custom theme.
2. Override less file defined in a parent theme/core module using a custom theme.
3. Use custom module's less files to customize core's less/add new less instructions.
4. Add third-party CSS file to a custom or a core page.

Before we discuss these situations in detail, it is important to understand when it is better to use module's less versus one defined in a custom theme. There is no strict rule that defines where to locate your less files. But the general principles are the following:

- Everything that is related to a theme, which is current design, should be located in a custom theme.
- Styles that are specific to a certain feature implemented in a custom module and are not likely to change, in case of redesign, should be located in a module's less files.
- Clearly, if there are no custom themes, or a module is to be distributed separately, the styles should belong to a module.

Customize less files using a custom theme

The main tool to customize less files without rewriting them is the use of the `web/css/source/_extend.less` file. This file is automatically included into the resulting css for every theme. Usually `_extend.less` includes other less files.

Further reading:

- Simple Ways to Customize a Theme's Styles

Override less files using a custom theme

In order to override less files, we should use the fallback mechanism. In Magento world, “fallback” is used to describe the priority in which Magento loads the file. The reason why fallback is needed, is the fact that the “same” file may be located in multiple places. For example, a phtml template can be located in the module’s folder, theme’s folder, and child theme’s folder. So Magento has to choose one file out of the three possibilities.

See DevDocs - Theme Inheritance for more details about the fallback mechanism.

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

Use custom module’s less files

Every custom module may include the `view/frontend/web/css/source/_module.less` file which is automatically included in the final css. This file usually contains either less instructions themselves or includes other less files.

Add third-party css file to a custom or a core page

Create `default_head_blocks.xml` layout file which includes:

```
<page xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xsi:noNamespaceSchemaLocat
    <head>
        <css src="css/custom.css" rel="stylesheet" type="text/css"  />
    </head>
</page>
```

Where `custom.css` is located at `view/frontend/web/css/custom.css`.

Further reading:

- Include CSS

4.04 Describe theme structure

Points to remember:

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

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

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

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

Describe the different folders of a theme

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

`i18n/` : this folder contains translations for the theme (`Magento/luma/i18n/en_US.csv`).

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

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

- `css/` : location of base Magento stylesheets. These will be exported to the `pub/static/[area]/[package]/[theme]/[locale]/css` directory.
- `css/source/` : LESS files that implement styles for basic UI elements. Most of these styles are mixins for global elements from the Magento UI library. `theme.less` is also located here, which overrides values for the default variables.
- `fonts/` : web fonts which will be utilized in the theme.
- `images/` : images that are included in the theme. These are images that will not frequently change. For example, you would include an icon here but not a free shipping banner.

- `js/` : theme-specific JS.

Module overrides:

When developing a theme, you will likely need to override another module's assets. These overrides reside in the theme folder, then the module's name. For example, in our SwiftOtter_Flex theme, we need to override `addtocart.phtml` in `Magento_Catalog` (`Magento/Catalog/view/frontend/templates/product/view/addtocart.phtml`).

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

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

Further reading:

- Theme Structure

Describe the different files of a theme

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

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

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

Examples:

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

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

Further reading:

- Theme Structure

4.05 Given a scenario, work with JavaScript files (basic)

JavaScript in Magento 2 is organized in modules which are loaded using RequireJs. See RequireJs in Commerce for more details.

Each module is a .js file located at the `view/(frontend|adminhtml|base)/web/js` folder. In order to call a module (for example to include it in the definition of another module, or execute using Magento js execution system) there is a special convention: `Module_Name/path/to/file`

For example: `Magento_Catalog/js/product/view/product-info` corresponds to the `view/frontend/web/js/product/view/product-info.js` file of the `Magento_Catalog` module.

It is also possible to give an alias to a js module. For this purpose another special file `view/(frontend|adminhtml|base)/requirejs-config.js` is used. For example:

```
var config = {
    map: {
        '*': {
            compareList:            'Magento_Catalog/js/list',
            relatedProducts:        'Magento_Catalog/js/related-products',
            upsellProducts:         'Magento_Catalog/js/upsell-products',
            productListToolbarForm: 'Magento_Catalog/js/product/list/toolbar',
            catalogGallery:         'Magento_Catalog/js/gallery',
            catalogAddToCart:       'Magento_Catalog/js/catalog-add-to-cart'
        }
    },
```

That was a snippet from `Magento_Catalog`'s requirejs config file. It tells us that `compareList` is an alias for `Magento_Catalog/js/list`.

Types of JavaScript modules in Magento 2

Plain modules:

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

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

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

jQuery UI widgets

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

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

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

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

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

UiComponents

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

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

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

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

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

Note that there are a number of different modules that extend `uiElement`. `uiCollection` is a common one and, as the name implies, facilitates dynamic lists. All UI Component JavaScript modules follow the basic pattern of extending a base class with an object containing a

`defaults` object and a number of functions. For one thing, the base class handles the template. This is why the methods (and properties of the `defaults` object) can be called from the template. We'll cover this in more detail later because it's complex.

Executing JavaScript modules

Using the

```
data-mage-init='{ "SwiftOtter_Module/js/modal": {“configuration-value”:true} }'
```

attribute. This special Magento attribute requests and instantiates a JavaScript module that takes two parameters: the element which hosts this attribute and the associated configuration (seen above in the ellipses). The module would look like:

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

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

Using the Magento script tag:

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

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

Imperative notation:

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

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

This is not recommended by Magento.

Customizing native JavaScript

In Magento 2, there are two instruments to customize a native js module. First is overriding using the fallback mechanism and second is the use of a mixin. Mixins are to JavaScript as plugins are to PHP in Magento. The logic is that you can add actions before, after, or instead of core functions. This allows you to manipulate core JavaScript without always needing to override the entire file. The benefit is you have to write and maintain less code. They are quite easy to use as long as the core is facilitating. There are some places where things get weird, but they can still be used.

Further reading:

- JavaScript Mixins

4.06 Describe front-end usage of customer data

Customer data in Magento means anything that is related to user-specific data that may affect full page caching. Usually, we use sessions to store some important user data that has to be rendered, however it brings a problem when a page is cached. Assume for example the following snippet from some phtml template:

```
Welcome <?php echo $this->getSession()->getUsername(); ?>!
```

This template may be included in all pages, including the homepage, product pages, and so on. Since a page's content depends on the session, it is impossible to cache a page's HTML, using FullPage Cache or Varnish. The solution to this problem is to move the data to a client's side (localStorage to be precise) and use JavaScript to render such pieces of HTML. This approach allows a page to be fully cached with some placeholders to be populated by a JavaScript module that is using data from a localStorage.

Customer data framework is a tool that provides a unified approach to this problem. See Page Caching - Private Content for more details.

In order to use the customer data module in JavaScript one has to

- Import the module via: `define(['Magento_Customer/js/customer-data', …], function(customerData, ..) {...});`
- Get the data via customerData.get method.

See `Magento/Checkout/view/frontend/web/js/view/minicart.js` as a canonical example for a customer data usage.

Setting the data and invalidation

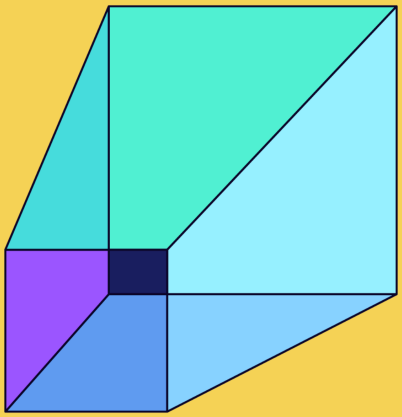
The big problem related to the customer data framework is invalidation. How do we change

the cached value (in `localStorage`) once it has changed on the backend? For doing so, Magento has a so-called `sections` mechanism. The main idea behind sections, is that each piece of data belongs to some "section" and has to be loaded/invalidated automatically. So, the sections mechanism consists of four parts:

1. Configuration, etc/section.xml file that describes what sections are present, and what actions may invalidate the section's content.
2. Frontend mechanism that sends AJAX request to load the sections data, once it has expired, or is missing.
3. Backend mechanism that pulls all the data from different places and sends it to the frontend call described in 1.
4. JavaScript event listener that reacts to any `POST` / `PUT` request and causes corresponding section invalidation.

Further reading:

- StackExchange: How do Customer Sections/Sections.xml work?



Checkout and Sales

Having a high-level view of how sales are processed.

Study Guide for Professional Developer Certification, AD0-E711

SwiftOtter

5.01 Describe cart components

Cart components from the end-user standpoint.

A user interacting with the website faces the following cart components:

- **Shopping cart:** It is activated when a product is added to the cart. In fact, adding a product to the cart is the most important operation of "shopping cart functionality". This functionality is, however, not as obvious as it may seem to be. There are multiple ways for a product to appear in a shopping cart, and users do not always realize it.
    1. **Adding a product using the "Add to cart" button on product/category pages.** This is the most straightforward way.
    2. **Reorder button in MyAccount.** It will try to add previously ordered products to a cart again, which may be a problem, since products might change over time. This is why it is important to keep this possibility in mind when planning a new implementation.
    3. **Wishlist**, gives another option of adding product to a cart. This is quite straightforward too.
    4. **Merge quotes functionality.** This is probably the most confusing of all. Quote merge is triggered when a customer visits a website, adds something to a cart, and then logs in. While logging in, Magento pulls the old cart and merges it with a new one. Both developers and business analysts often neglect this scenario which is causing bugs that are difficult to identify.
- **Cart items:** A user can see their added to cart items on the shopping cart page, or using the mini cart icon in the top right corner of every page (in the Luma theme layout).
- **Shipping address:** Users are facing a "shipping address component" when they enter their zip code and shipping address to estimate a quote on the shopping cart page. This triggers a shipping price calculation process that is usually initiated during a checkout.
- **Discounts:** A user has an ability to add a coupon code in the shopping cart to receive

a discount. This is managed by Promo Rules in the admin panel.

- **Totals:** User is seeing totals in the minicart and on the shopping cart page. Usually they see subtotal, discount, tax.

Cart components from a developer's standpoint

While in the previous section we described a "shopping cart functionality", here we describe underlying classes and typical operations/customizations related to that functionality.

Shopping cart:

The cart itself is represented by two classes: `Magento\Checkout\Model\Cart` and `Magento\Quote\Model\Quote`. The first one can be viewed as a wrapper on the second, while the `Quote` class is the key class that performs all the operations. Typically the `Quote` class is responsible for:

- Adding product to a cart. Note, that `Quote` is a common denominator for all four scenarios described above.
- Initiating totals recollection. It is a `Quote` that decides when it is time to recollect totals. Note, that `Quote` itself does not own the total models.
- Owning items and addresses (billing and shipping). `Quote` is responsible for all operations with items, such as add, delete, update. Another important aspect of the `Quote`'s class is deciding when an item has to be added as a separate line item, or when it should be "included" in another existing item by increasing its quantity. A typical example is adding two instances of the same product with different custom options.
- Owning a `Payment` object (this is checkout rather than shopping cart functionality).

For more information see the source code: `Magento\Quote\Model\Quote`. Pay attention to methods:

- `beforeSave()`
- `*load()`
- `assignCustomer()`
- `getBillingAddress()`
- `getShippingAddress()`
- `getAllShippingAddresses()` (when multishipping checkout is enabled, a quote may have more than one shipping address)
- `getAllItems()`
- `getAllVisibleItems()`
- `addItem()`
- `addProduct()`
- `updateItem()`
- `getIsVirtual()`
- `merge()`
- `collectTotals()`

Cart Items

An item itself is represented by the Magento\Quote\Model\Quote\Item class. It is a relatively simple class. Note that items are actually owned by a `Quote Address` not `Quote`, although `Quote` allows fetching and adding items.

Very important methods are:

- `representProduct()` - it defines whether two items are to be joined (with increased qty) or separated.
- `setProduct()` - it defines which product's attributes are available (see 5.04 about discussing quote's data).

Another important aspect of Items functionality is items rendering on a shopping cart page. This is done by a special Renderers which could be configured via layout xml for different product types. Here is an example for a simple product (`Magento/Checkout/view/frontend/layout/checkout_cart_item_renderers.xml`):

```
<referenceBlock name="checkout.cart.item.renderers">
    <block class="Magento\Checkout\Block\Cart\Item\Renderer"
        name="checkout.cart.item.renderers.default" as="default"
        template="Magento_Checkout::cart/item/default.phtml">
        <block class="Magento\Checkout\Block\Cart\Item\Renderer\Actions"
            name="checkout.cart.item.renderers.default.actions"
            as="actions">
               <block
                  class="Magento\Checkout\Block\Cart\Item\Renderer\Actions\Edit"
                  name="checkout.cart.item.renderers.default.actions.edit"
                  template="Magento_Checkout::cart/item/renderer/actions/edit.phtml"/>
               <block class="Magento\Checkout\Block\Cart\Item\Renderer\Actions\Remove"
                name="checkout.cart.item.renderers.default.actions.remove"
                template="Magento_Checkout::cart/item/renderer/actions/remove.phtml"/>
        </block>
    </block>
```

Finally it is worth discussing how different product types are represented item wise. So, a simple product corresponds to a single item, configurable product corresponds to two items: configurable sku (parent) + selected sku (child, “real” item), bundle will be represented as a bundle sku + one sku per selection, grouped is represented by a set of selected skus without indicating any relationship to a grouped product. You may actually have more item records in a database then displayed on a shopping cart page. This is also a reason of many parent/child checks that you may find in various places in the core code related to operations with items.

Shipping Address

Both Shipping and Billing addresses are represented by the

`Magento\Quote\Model\Quote\Address` class. This class also holds items—physical items to be delivered. Virtual items belong to a billing address. In case of multishipping checkout, there could be more than one shipping address for a quote. Another important responsibility of the `Address` class is to collect shipping rates. See 5.05 for shipping methods and rates discussion.

In particular, pay attention to these methods:

- `getAllItems()`
- `getAllVisibleItems()`
- `getShippingRatesCollection()`
- `requestShippingRates()`

Discounts

See 5.02.

Totals

When calculating totals for a shopping cart/order, such as — Subtotal, Tax, Discount, Shipping, Grand Total and others, Magento uses a framework of Total models. Total models are registered in `etc/sales.xml` (`Magento/Quote/etc/sales.xml`).

You can easily add another total model. A total model does two things—calculates its own total amount and defines which data are needed for rendering (done by JavaScript). See for example `\Magento\Quote\Model\Quote\Address\Total\Subtotal`.

The `collect()` method for totals calculation (pay attention to `Magento\Quote\Model\Address\Total` parameter, it is a container of total amounts) and the `fetch()` method returns data for rendering.

5.02 Describe a cart promo rule

Promo rules are technically applied via the Total models framework (see 5.01 and `Magento/SalesRule/etc/sales.xml`).

The structure of a Promo Rule is similar to that of Catalog Rule (both use the same Rule engine). The only important difference is the ability to create and configure coupon codes.

Two core elements of a Promo rule are—Conditions and Actions.

Conditions is a tool that allows you to configure a boolean function that defines whether a rule should be applied for a cart or not. One of the most important aspects of Promo Rules Conditions is the ability to use EAV attributes to trigger a rule. In order to enable an attribute to be used for Promo Rule, its property "Used in Promo Rule Conditions" should be set to true. See StackExchange - Shopping Cart Rule Based on Custom Attribute.

There are four possible actions which define how a rule should be applied. The documentation says:

- Percent of product price discount
    - Discounts item by subtracting a percentage from the original price. The discount applies to each qualifying item in the cart. For example: Enter 10 in Discount Amount for an updated price that is 10% less than the original price.
- Fixed amount discount
    - Discounts item by subtracting a fixed amount from the original price of each qualifying item in the cart. For example: Enter 10 in Discount Amount for an updated price that is $10 less than the original price.
- Fixed amount discount for whole cart
    - Discounts the entire cart by subtracting a fixed amount from the cart total. For example: Enter 10 in Discount Amount to subtract $10 from the cart total. By default, the discount applies only to the cart subtotal. To apply the discount to the subtotal and shipping separately, use the Apply to Shipping Amount option.

- Buy X get Y free
    - Defines a quantity that the customer must purchase to receive a quantity for free. (The Discount Amount is Y.)

It is worth mentioning how discounts are related to shipping cost. A rule can be configured so that it discounts shipping or not. It is also possible to configure a “Free shipping” discount. The important aspect though, is that it is the job of the corresponding carrier (Shipping method) to decide how to apply the discount or free shipping. So whenever you work on a new shipping method you should process discounts when calculating shipping rates.

Note: there is a subtlety in calculating discounts together with taxes (see Tax Calculation Settings).

Further reading:

- Cart Price Rules

5.03 Given a scenario, describe basic checkout modifications

See 5.01 for Shopping Cart discussion. The classes described in 5.01 play a vital role on checkout as well. Other than that, the checkout modifications could be reduced to two functional areas: checkout flow (steps) and order placement.

Checkout Flow

Checkout process consists of two steps—shipping and billing. Shipping step includes shipping address and shipping methods substeps. Billing includes billing address definition, payment method selection, and submitting and order.

Checkout steps are defined in the `view/frontend/layout/checkout_index_index.xml` layout file. Note that the interface itself is rendered by a group of UiComponents, which configuration is coming from that layout xml file. This approach is generally speaking, **unusual**,

and Magento does not support UiComponents configuration in layout xml out of the box. There is a lot of backend code that parses layout xml instructions and converts them into the UiComponents configuration.

To get yourself familiar with the JavaScript framework that renders checkout steps, see `Magento/Checkout/view/frontend/web/js/`.

While the whole process is complicated, there are some general rules to remember when customizing checkout steps:

- Most of the work is done by JavaScript, so we use mixins to customize them.
- Layout xml configuration is flexible, and sometimes it is possible to change a component that does something to a custom component in layout xml.
- JavaScript framework that implements checkout can be tentatively seen as the following MVC paradigm—there are "actions" that send requests to a backend (`Magento/Checkout/view/frontend/web/js/action/`), models that are responsible for data storage and some business logic (`Magento/Checkout/view/frontend/web/js/model/`), and views that are responsible for rendering (`Magento/Checkout/view/frontend/web/js/view/`). This is tentative, because it is not always possible to split responsibilities according to MVC, plus a controller role is somewhat reverse: actions are used to send the data, rather than receiving a request in a classical MVC, plus the physical request is often executed by a model not action (so action may be preparing data and calling a model, like in `Magento/Checkout/view/frontend/web/js/action/place-order.js`). But this logic helps to identify the right module and the right approach to a customization. So, if all you need is to change a UI, then you probably should be looking into the `view` folder. If it has anything to do with the data being sent to the backend, that must be somewhere in `actions` and `models`. Despite all its complexity, JavaScript is only responsible for the front side of the matter (although broken JavaScript effectively causes inability to complete checkout which is always a severe issue). The main work

is done on the backend. JavaScript modules (mainly `actions`) send REST requests to the backend, using the standard Magento Web API. Magento uses a special `self` authentication mechanism to signal that the request should only be applied to a given (frontend!) session.

See `Magento/Checkout/etc/webapi.xml` for a list of REST resources that could be used during the checkout.

Order Placement

Order placement is a final operation in a checkout process. It can be seen as a sequence of steps:

1. Convert quote information to order (which means creating order, order items, order payment, order addresses, objects and populate them with data). This is done with a help of converter classes (see `\Magento\Quote\Model\QuoteManagement`, the `submitQuote()` method).
2. Place an order, which means to submit a payment (see `\Magento\Sales\Model\Order` and `\Magento\Sales\Model\Order\Payment`).
3. Save an order, invalidate quote, send configuration email (which could be done asynchronously by a cron job). Sometimes, Magento will also create an invoice.

Note, the very useful event `checkout_submit_all_after`, which is fired after the order's transaction has been committed, so that your code, once broken won't affect the payment and saving order process. Usually, this event is used extensively for all custom operations that have to be executed after an order has been placed.

Further reading:

- Customize Checkout

5.04 Given a scenario, describe basic usage of quote data

In 5.01 and 5.03, we discussed key aspects of the shopping cart and checkout functionality, involved classes, and their responsibility. In this section, we discuss how we can obtain or manipulate data related to a shopping cart.

Quote

In order to obtain a quote object, we usually use `\Magento\Checkout\Model\Session::getQuote()` method. Please check the source code, since there are some subtleties related to an initial quote loading process. Also, refer to 5.01, the discussion of a quote merging.

Quote Addresses

First of all, it is important to understand how many addresses there could be. If Multishipping checkout is enabled, there could be many shipping addresses and one billing address. If Multishipping is disabled, there is one billing address and at most one shipping. There could be no shipping addresses if the quote is virtual. In order to operate with addresses use `\Magento\Quote\Model\Quote`. It has a group of addresses-related methods. See lines 1143-1400.

Total amounts

Use `Address` object, `getTotals()` method to obtain a list of calculated total amounts. See 5.01, Totals discussion.

Items

Getting a list of items can be tricky, since items belong to both `Quote` and `Address` objects. In 95% of all situations, you would use a `Quote` object to fetch them. See a quote class source code, roughly lines 1400-1600 for item-related methods. Please mind the difference

between `getAllItems()`, `getAllVisibleItems()`, `getItemsCollection()` (refer to 5.01, Items discussion for parent/child relationship). Also note, that an item may be deleted but still remain in a collection.

Items options

Some important information does not fit into the model described above. This includes information about product's custom options, some technical information regarding product types (like bundles), and so on. Such information usually goes into item's options, which are stored in the `quote_item_option` table.

The ability to store options on the item level could also be very important for customizations. Often it is very simple to store additional pieces of information in an item's option rather than in a custom table that is related to an item, which requires much more work to develop and maintain.

Product's attributes available in item

It is a typical operation to fetch an instance of a product from a given item. However, in some situations this instance may not include all of the product's EAV attributes. For example, it is possible that a customer will have many items in a cart, so loading all attributes for all items could be costly. Instead Magento only loads some of the attributes, when loading items collection:

```
$productCollection =
    $this->_productCollectionFactory->create()
    ->setStoreId($this->getStoreId())
    ->addIdFilter($this->_productIds)
    ->addAttributeToSelect($this->_quoteConfig->getProductAttributes());
```

See `\Magento\Quote\Model\ResourceModel\Quote\Item\Collection` for more details.

The list of attributes to load is defined in the `etc/catalog_attributes.xml` file, see

`Magento/Sales/etc/catalog_attributes.xml` as an example.

5.05 Given a scenario, configure the payment and shipping methods

Shipping method configuration

Shipping (delivery) methods could be online and offline. Offline methods, such as FlatRate, TableRate, Free Shipping and In Store delivery do not perform any online requests when calculating shipping rates. Typically for **offline methods**, one has to configure its name, price, and other relevant information specific to each method, such as: conditions to apply Free Shipping, Handling Fees, and so on. Note, that a TableRate method requires a special csv file with rate information to be uploaded in the admin panel (which is a great reference for uploading files in system configuration!). You can pull a reference file and fill it in with the relevant rates.

Online methods (also called Carriers) typically require credentials used to request their APIs during a checkout process to obtain shipping rates. Other than credentials, configuration may vary from method to method.

Further reading:

- Delivery Methods

Payment methods configuration

There are three types of payment methods in Magento: offline, gateway, and hosted.

Offline are those which do nothing during checkout; they are the simplest one. Usually they assume that a payment will be executed outside of Magento (by sending a check or a bank wire). Offline methods are: Check/Money order, Bank transfer, Zero subtotal checkout, Cash on delivery and Purchase order. Configuration of these methods is pretty straightforward. See: Payment Methods.

Gateway and hosted methods

Gateway method is one that sends a remote request to a payment provider to place a payment. Hosted is one that redirects customers to another website, where they complete their payment and then redirect back (example is PayPal Express checkout). Note, that in 2.4, Magento removed all online methods except PayPal from the core product. For such methods one usually has to configure credentials needed for an API request/Redirect and Payment Action. Payment Action describes what Magento should do when placing a payment. They are configured by payment method so may differ. However, there are three principal options supported by the `\Magento\Sales\Model\Order\Payment` class out of the box— `Authorize only`, `Capture` and `Order`. Other actions will be delegated to a payment method class rather than the `Payment` class.

`Authorize only` — will only authorize funds on a card without actually charging them. Invoice will not be created. An admin must manually hit the "Invoice" button when managing an order, which will effectively execute the capture operation and charge a card.

`Capture` — will charge the card. Magento automatically creates an invoice, no action needed for an admin.

`Order` — does nothing, invoice is not created. An admin must charge the card when creating an invoice (but nothing guarantees that the amount is available).

Usually online methods use tokens to perform consequent operations (like authorize, then capture then refund). However, exact details depend on a method's implementation.

See the DevDocs for configuring PayPal Express. Please note the "Enable In Context" checkout option. It is aimed to improve a user's experience so that a user doesn't feel like being redirected to another website, although the functionality is pretty much the same.

Further reading:

- Other PayPal Methods

5.06 Given a scenario, configure tax rules, currencies, cart, and/or checkout

Configure tax rules

In order to configure tax rules one has to configure:

1. Product tax classes (a special tag that is used to group products for taxation, see Configuring Tax Classes).
2. Customer tax classes (a special tag that allows tweaking tax calculation for different customer groups, Configuring Tax Classes).
3. Tax rates (zones). Tax rate defines a physical location (state, city, range of zip codes) to which a given tax rate has to be applied (Tax Zones and Rates).
4. Tax rules themselves. Finally, tax rules brings it all together and connects a product tax class, customer tax class, and a rate. See Tax Rules.

Configure currencies

There are two primary elements of configuration related to currencies—currency scope and store's currencies.

Currency scope (also referred as a price scope) defines what is a scope for a base currency. It could be global or a website. Effectively it defines how many different prices a product may have (options are one global price or one price per website).

Store's or display currency defines a currency that a customer sees. It has a store scope (Store in developer's terminology and Store View in merchant's). Display currency calculated based on currency rates applied to a base currency.

Further reading:

- Currency Configuration
- Currency Setup

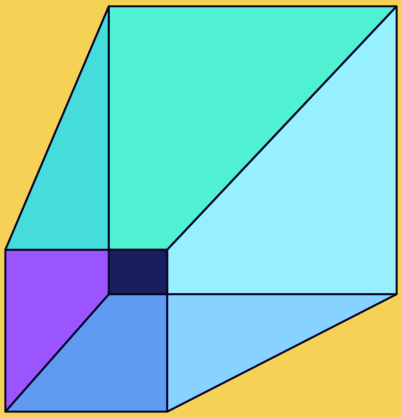
Configure shopping cart/checkout

There are a bunch of configuration options that allow you to modify the checkout look and feel as well some behavioral aspects. The most important are:

- Enable Onepage checkout
- Enable Multishipping checkout
- Allow guest checkout
- Display mini cart

Further reading:

- DevDocs: Checkout



Catalog

Understanding products, categories and price modifications.

Study Guide for Professional Developer Certification, AD0-E711

SwiftOtter

6.01 Identify the basics of category management and products management

See 6.02 for a general discussion about products and product types. See 6.04 for product prices discussion. See 3.05 and 3.02 for a discussion about product's attributes. See Product Settings for general information about product settings and inventory, and Catalog Images and Video for images.

Next we will discuss categories and product-to-category relationship.

How do you create and manage categories?

To create or manage a category, the first place we look is the Category data interface (`\Magento\Catalog\Api\Data\CategoryInterface`). This represents Magento's contract to us of what is unchangeable throughout releases (until Magento version 3). The implementation for the Category interface is the Category model (`\Magento\Catalog\Model\Category`).

While no errors will stop you from writing your application against an implementation of a data interface, you run the risk of incompatibilities for future versions of Magento. The safest is to always use data interfaces. For example, you would write a method like this:

```
public function getCategoryPath(CategoryInterface $category): string
{
    return $category->getPath();
}
```

But what if a method is not available in the data interface? For example, the Category model has a method, `getUrl()` but the data interface does not have this method. What to do?

Write your code against the implementation of the interface: for example the `Category`

model instead of the `CategoryInterface`. You can even make it more flexible like:

```
public function getCategoryUrl(CategoryInterface $category): string
{
    if (!($category instanceof \Magento\Catalog\Model\Category)) {
        throw new \InvalidArgumentException('Category must be a Category model.');
    }

    return $category->getUrl();
}
```

Most developers omit the exception with the assumption that the implementation of the `CategoryInterface` is always the `Category` model.

Another variant of the above is to change the `CategoryInterface` parameter type to be the implementation type, the `Category` model. The net result of the example and this is the same: an exception is thrown. However, in the above example, you have more control over what happens (for example, you don't have to throw an exception, and can silently fail).

Further reading (important):

- DevDocs: Service Contracts
- DevDocs: Service Interfaces
- Vinai Kopp: Repositories, Interfaces and the Web API
- BelVG: Service Contracts

Creating a category object will happen with a factory. Remember, as we discussed in Objective 1, you have two options for the factory:

- `\Magento\Catalog\Api\Data\CategoryInterfaceFactory`: this one takes the result of the `preference` for this interface and creates a factory for it. Ultimately, this ends up being: `\Magento\Catalog\Model\CategoryFactory`

Once you have a category object, you can set any applicable values on it.

To save it, use the `CategoryRepositoryInterface` and call the `save` method.

Practical experience:

- Set a breakpoint in the repository's `save` method.
- Create a category, programmatically.
- Step through the `save` method.

Example (using a data patch):

```php
<?php
namespace SwiftOtter\OrderExport\Setup\Patch\Data;
// app/code/SwiftOtter/OrderExport/Setup/Patch/Data/CreateCategory.php

use Magento\Catalog\Api\CategoryRepositoryInterface;
use Magento\Catalog\Api\Data\CategoryInterfaceFactory;
use Magento\Framework\Setup\ModuleDataSetupInterface;
use Magento\Framework\Setup\Patch\DataPatchInterface;

class CreateCategory implements DataPatchInterface
{
    /** @var ModuleDataSetupInterface */
    private $moduleDataSetup;

    /** @var CategoryRepositoryInterface */
    private $categoryRepository;

    /** @var CategoryInterfaceFactory */
    private $categoryFactory;

    public function __construct(
        ModuleDataSetupInterface $moduleDataSetup,
        CategoryRepositoryInterface $categoryRepository,
        CategoryFactory $categoryFactory
    ) {
        $this->moduleDataSetup = $moduleDataSetup;
        $this->categoryRepository = $categoryRepository;
        $this->categoryFactory = $categoryFactory;
    }

    public function apply()
    {
        $category = $this->categoryFactory->create();
        $category->setName('My Category');
        $category->setIsActive(true);
        $category->setParentId(2);

        $this->categoryRepository->save($category);
```

Note: If you copy the above code into your IDE, you will get a warning that `Magento\Catalog\Api\Data\CategoryInterfaceFactory` is not a valid class. However, proceed to execute the code, and you will see the error will disappear. How does this happen? When PHP tries to autoload this class, it reaches out to Composer (`\Composer\Autoload\ClassLoader::loadClass()`). This happens thanks to the `spl_autoload_register` method. Magento also registers its own autoloader class (`\Magento\Framework\Code\Generator\Autoloader`). When this method is called, Magento passes the "request" off to the `Generator` (`\Magento\Framework\Code\Generator`).

How do you assign and unassign products to categories?

That's a great question! Let's take the opportunity to dig in and understand how to find the answer. The first question I ask when building something in Magento is "is this already done" or "do I have an example of something similar in Magento?" The answer to both of these questions is "yes."

First, look in the admin area. Of course, we can add/remove products from a category. We would look for the Category save controller (`\Magento\Catalog\Controller\Adminhtml\Category\Save`). We can see in the `execute()` method some code to `setPostedProducts()` on the category. I would then search for the reverse `getPostedProducts()` in `vendor/magento/module-catalog` . This will dig up the Category resource model (`\Magento\Catalog\Model\ResourceModel\Category`). You will find a method named `_saveCategoryProducts()` . In PHPStorm, you can right click on the method name and select Find Usages (on Mac, this is Alt+F7). This method is called in that Resource model's `_afterSave()` method.

In other words:

- Load a category.
- To get the original product assignment list, call the category's

`getProductsPosition()` method.

- Call the `setPostedProducts()` magic method on the category. Pass a key-value array with the key being a product ID and the value being the sort order.
- Save the category (note that if your indexer is set to Update on Schedule, and your cron is running, it could be up to an hour before the associations are made).

Another place to find the most functionality is through the API. You can search through the API documentation for category, then look for where an API call mentions **both** a category and a product (Admin REST API Docs).

Search in the `vendor/magento` directory for the request URL (you will need to convert `{categoryId}` to be `:categoryId`.

API Request URL

This leads you to a place that documents exactly the process listed above: `\Magento\Catalog\Model\CategoryLinkRepository::save()`.

A couple of other thoughts:

When category/product associations are changed, if the Index type is set to Update on Schedule, at the time of saving, the `catalog_category_product` index is marked as needing an update (see `\Magento\Catalog\Observer\CategoryProductIndexer` and this one `\Magento\Elasticsearch\Observer\CategoryProductIndexer`). The indexer is called from the `\Magento\Catalog\Model\Category` class.

A category's product URLs are created when the category is saved. Set a breakpoint here: `\Magento\CatalogUrlRewrite\Observer\CategoryProcessUrlRewriteSavingObserver`. Step through this code to get a better picture of how this happens.

- This class fetches a list of URL rewrites from the `UrlRewriteHandler`.
- `UrlRewriteHandler` iterates through the changed products and gets the rewrites from `ProductUrlRewriteGenerator`.
- Traversal continues to generate for a specific store view (`\Magento\CatalogUrlRewrite\Model\ProductScopeRewriteGenerator`).
- Generators:
    - `CanonicalUrlRewriteGenerator`
    - `CurrentUrlRewritesRegenerator`, which has a number of generators.

6.02 Describe product types

Product overview

A core feature of Magento is to present products online that are for sale. A customer selects which products they wish to purchase and those products are added to the cart. These online representations are mapped to a virtual product (like an ebook) or physical products sitting on shelves at a merchant and are shipped to the customer.

If you noticed, I used the word “mapped” when talking about online to physical representations. Magento provides powerful tools to help the customer decide which product they wish to order. But these tools do not get in the way of determining which product(s) to pull from the shelves and ship to the customer.

We will now review each type of product, present use cases for each, and some additional details. This area of the test is less focused on code but more on the Admin implementation of these products. You, as a developer, must have a good grasp on what product type to use where, and what the implications of this decision are.

Simple

A simple product should be a 1:1 mapping of a product in Magento to an item on the shelf. A simple product is the basic unit of inventory.

The package of 10 pens in Magento represents the box on the merchant's shelf that will be shipped to the customer: when they open the package of 10 pens, they will happily find 10 pens. Note that this product type would not work if pens are sold individually, but in Magento, the merchant virtually groups these as a package. Why? Because there is no default capacity to map inventory (purchasing 1 of this product results in 10 of these products shipping). In this example, it does work, because the merchant ships 10 pens as one unit.

A small, red t-shirt in Magento (that will be included in a configurable "select your size and color" product) means a small, red t-shirt on the merchant's shelf.

A 64GB stick of computer memory in Magento (that will be included in computer bundle product).

A simple product can be a child (included) in:

- Configurable products
- Bundle products
- Grouped products

However, a simple product cannot be a child if the simple product has custom options. The confusing part is that you can associate this product in the admin panel, but the product will not appear on the frontend. For reference, custom options are adding when editing a product:

Simple Product as Customizable Option

Virtual

Virtual products are similar to simple products except that they do not represent a unit of physical inventory. Virtual products represent non-tangible goods: a subscription to a magazine, a gym membership, or possibly a gift card (though, Magento Commerce natively includes this functionality).

Using the above use cases, you could create a bundle product to sell fitness equipment, then the customer can add a fitness coaching plan.

Configurable

The best way to think of a configurable product is a tool to filter down a list of products to one product. The store administrator will:

- Create the configurable product.
- Assign product attributes (that are global and dropdown or swatch).
- Create or assign products that have a value specified for each product attribute assigned to the parent configurable product.

At this point, we have a list of simple products assigned to our configurable product (creating what I call a parent-child relationship).

You can see an example of this list of child products here by going to the admin panel > Catalog > Products > (click the Filter button) > Type: Configurable.

Configurable Product Children

Here is an example of how this is rendered:

Configurable Product Rendering

When the user selects “S” (small) for size and the color, Blue, then clicks Add to Cart button, these items are added (see the `quote_item` table):

Configurable Product quote_item table

The first row represents the parent configurable product. Notice the “2” quantity for the parent configurable product and the “1” quantity for the child product. This means that there is 1 child product for each parent configurable. Ultimately, as we will see shortly, 2 children will be shipped as the number equals `[Quantity of Child] x [Quantity of Parent]`.

Once the order is placed, here are the values from the `order_item` table:

Configurable Product order_item table

Note that the `qty_ordered` reflects the number of products that are being shipped.

As you can see in this example, we have two attributes. These attributes help the customer easily select which simple product they would like. The items ordered include the configurable parent and the simple child, however, only the simple child is shipped.

What is the `base_` column prefixes?

One of the powerful features of Magento 2 is the ability to upload prices in one currency but convert these prices to a more relevant currency for the visitor.

The price in which a product is uploaded is called the base currency. This can be specified in whatever scope the product's price is marked as (see Stores > Configuration > Catalog > Pricing > Product Price Scope). In other words, if the product's price is assigned to the website scope, you can also set the base currency in the website scope.

The base currency is also the currency in which an order is charged. For example, if the base currency is USD (United States Dollars) and the converted currency is INR (Indian Rupees), the value sent to the payment processor would be the value in USD, but the customer would see INR on the frontend.

Here is an example of how this is shown in the admin panel:

Base Currency Display

The amount in brackets is the value shown to the customer. The amount above the bracketed amounts is the base currency.

The base currency is also ideally used for reporting. In the above example, if we based our report inputs on the amount shown to the customer, one order might have a value of 10, and another order might have a value of 686 (for the same product).

Grouped products

Grouped products allow you to display a list of similar products. This sounds similar to

configurable products EXCEPT that you can add multiple products to the cart at one time.

This is ideal for:

- sandpaper where you might want to add different roughnesses at the same time
- pipe fittings where you would want to select multiple sizes to be added

Typically speaking, a grouped product is used where the customer might want to add multiple similar products to the cart from one page.

Grouped Product Display

After clicking add to cart, here are the items that are added to the `quote_item` table:

Grouped Product quote_item table

Then, placing the order results in these rows in the `sales_order_item` table:

Grouped Product sales_order_item table

In this example, we must notice that the `product_type` does not reflect the actual product's type. Rather, it inherits the parent (which doesn't exist in either the `quote_item` or `sales_order_item`).

Bundle

The last product to review is the Bundle type. This is ideally used in situations where the customer can configure the product they want.

The best example would be configuring a computer: XYZ processor, 64GB of memory, a 1TB solid-state hard drive, etc.

Magento's demo store allows the visitor to configure a bundle of products. A bundle product can apply a discount to incentivize using the bundle instead of products individually.

Bundle Product Display

Once we add the above combination to the cart, we can find these entries in the `quote_item` table:

Bundle Product quote_item table

And, then, when the order is placed, we see these entries in `sales_order_item`:

Bundle Product sales_order_item table

How would you obtain a product of a specific type?

There are two ways: 1) use `SearchCriteriaBuilder` with a `ProductRepository` or 2) use a product collection with `addFieldToFilter('type_id', 'simple')`.

Using **`SearchCriteriaBuilder`**:

```
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 getAllGroupedProducts(): array
{
    $criteria = $this->searchCriteriaBuilder->addFilter(
        'type_id',
        \Magento\GroupedProduct\Model\Product\Type\Grouped::TYPE_CODE
    )->create();

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

Remember, you must call the `SearchCriteriaBuilder`'s `create()` method, otherwise, you will get a fatal error as `SearchCriteriaBuilder` does not inherit `SearchCriteriaInterface` and thus the types are not interchangeable.

Using a collection:

```
    private $collectionFactory;

    public function __construct(
        \Magento\Catalog\Model\ResourceModel\Product\CollectionFactory
            $collectionFactory
    ) {
        $this->collectionFactory = $collectionFactory;
    }

    public function getAllGroupedProducts(): array
    {
        $collection = $this->collectionFactory->create();
        $collection->addAttributeToSelect([
            'name',
            'url_key'
        ]);

        $collection->addFieldToFilter(
            'type_id',
            \Magento\GroupedProduct\Model\Product\Type\Grouped::TYPE_CODE
        );

        return $collection->getItems();
    }
```

The advantage of a collection is that only the attributes we specify are loaded. This reduces the overhead in obtaining these products.

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

There are two parts to a product: the product model (`\Magento\Catalog\Model\Product` which implements `ProductInterface`) and the product **type** model (for example, the `grouped` type model (`Magento\GroupedProduct\Model\Product\Type\Grouped`) inherits

`\Magento\Catalog\Model\Product\Type\AbstractType`).

The type model is available through the product's `getTypeInstance()` method.

Why do we need a product model and a product type model?

The product model is the doorway to the product. This model contains the data from the database that pertains to this product (the product's ID, SKU, name, etc.). The product's type model contains methods that apply to this product type, whether it be simple or grouped or configurable.

For example, the grouped type product's model has the ability to:

- `getChildrenIds()`: find the product IDs of all child products that are assigned to the parent grouped product.
- `getParentIdsByChild()`: this finds all parents associated with a particular child.
- `getAssociatedProducts()`: this loads all products associated with a parent.

I suggest you take a few minutes to review these files:

- `\Magento\Catalog\Model\Product\Type\AbstractType`
- `\Magento\Catalog\Model\Product\Type\Simple` (simple)
- `\Magento\Catalog\Model\Product\Type\Virtual` (virtual)
- `\Magento\Downloadable\Model\Product\Type` (downloadable)
- `\Magento\ConfigurableProduct\Model\Product\Type\Configurable` (configurable)
- `\Magento\Bundle\Model\Product\Type` (bundle)
- `\Magento\GroupedProduct\Model\Product\Type\Grouped` (grouped)

Further reading:

- Simple Product
- Configurable Product

- Grouped Product
- Virtual Product
- Bundle Product

Next we discuss product type's behavior in a shopping cart (see 5.01 for general discussion on Shopping Cart).

How are configurable and bundle products rendered?

Again, we will view this answer as more practical experience than reading.

To find the answer to this question:

- Locate the Sprite Yoga Companion Kit on the frontend (`/sprite-yoga-companion-kit.html` , by default).
- Add it to the cart.
- Go to the cart page.
- Highlight the item in the cart, to locate the corresponding elements in Chrome Developer Tools > Elements:

  Looking up element with dev tools

- Look for anything pertaining to a "bundle" in the element list:

  Finding bundle reference

- Unfortunately, we do not see anything, so let's start with something that might be fairly unique to a bundle or configurable product, the `item-options` display. Remember, anytime you can pre-guess the file extension, that will filter down your results.

  PhpStorm lookup with filter

- There are quite a few entries in this list. However, most of them pertain to the `order`

and not the `cart`. There is only one item on the list that pertains to the `cart`: `Magento/Checkout/view/frontend/templates/cart/item/default.phtml`. Let's see how this template is utilized in layout XML.

- Search for the template path (everything after `templates`) and filter by all `*.xml` files:

  Template path lookup with xml filter

- This is great! We have now found that `checkout_cart_item_renderers` layout handle is used to store cart item renderers.

- But how does the rendering class even work? Let's find out.

- Open
  `vendor/magento/module-bundle/view/frontend/layout/checkout_cart_item_renderers.xml`
  and locate the block that contains our renderer block:
  `checkout.cart.item.renderers`. We need to find where that block is declared.

- Search `vendor/magento` again for `"checkout.cart.item.renderers"`. Notice the quotes? That eliminates all entries like `checkout.cart.item.renderers.bundle`.

  Name lookup with quotes filter

- Now that we know the block which renders the cart items, we need to figure out how it actually works. Here is the excerpt from
  `Magento/Checkout/view/frontend/layout/checkout_cart_index.xml`:

```
<block class="Magento\Checkout\Block\Cart\Grid" name="checkout.cart.form" as="ca
    <block class="Magento\Framework\View\Element\RendererList" name="checkout.ca
    <block class="Magento\Framework\View\Element\Text\ListText" name="checkout.c
</block>
```

- Let's look at `Magento/Checkout/view/frontend/templates/cart/form.phtml`. In here, we see the call to `<?= $block->getItemHtml($_item) ?>`.
- `getItemHtml()` loads the renderer block, which calls `getRenderer()`.
- Ultimately, the renderer just returns the block whose `as` attribute matches the methods `type` parameter, which will be the type of the product being rendered (`bundle` for example).

How can you create a custom shopping cart renderer?

Now armed with this new knowledge, we can see how to create a cart item renderer for a new product type. You can also override or create a new renderer for an existing type with basically the same steps:

- Create an XML file in `app/code/[vendor]/[module]/view/frontend/layout/checkout_cart_item_renderers.xml`.
- Reference the block `checkout.cart.item.renderers`.
- Create a new block with the `as` matching the type of the product (`kit` as an example of a new product type).
- Set the `template` attribute. Most use the default `Magento_Checkout::cart/item/default.phtml`, but you can use whatever template you wish.

- The block type should extend `\Magento\Checkout\Block\Cart\Item\Renderer`. See the bundle renderer (`\Magento\Bundle\Block\Checkout\Cart\Item\Renderer`) as an example.
- Presto!

6.03 Describe 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`

6.04 Describe price types

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 (i.e. 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 `Magento/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:

```
Magento/Downloadable/view/frontend/layout/
catalog_product_view_type_downloadable.xml
```

The templates for these renderers are found here: `Magento/Catalog/view/base/templates/product/` .

Additionally, there is a JS UI component to display prices (Render Prices on the Frontend). These templates can be found in `Magento/Catalog/view/base/web/template/` .