Associate Developer Project Guide

Build an order export tool to push data into a custom ERP

Joseph Maxwell

Dear reader,

I hope you find this guide helpful. This is designed as a companion for the Associate Developer study guide. Throughout this guide, I will provide references to where you can learn more in the study guide.

The difference between the study guide and this project guide is:

- The study guide covers every topic on the test, in-depth.
- This guide walks you through creating a project.
- The examples in the study guide are not related, but focused on the specific discussion for that topic.
- The example in this guide is one cohesive unit, but doesn't have as much in-depth information for each subject.

Of course, I would suggest that these are used in tandem. That is up to you.

The goal of this project guide is this: I wish I had the time to sit down with you and help you build this module. Alas, but that is not a possibility. So this is the next-best thing.

I love to teach and share the knowledge that others have so kindly helped me out with. I pass on that knowledge here.

I estimate that this study project covers over 70% of the knowledge required to pass the test. There are a few areas that are not covered at this moment:

- 1.6: Configure event observers and scheduled [cron] jobs. I recommend that you add an event observer to this project and follow the path of how this works.
- 3.3: We do not utilize block types other than the default template. A thorough review of this would be beneficial.
- 5.4: Set up a menu item.
- 6: Product types, categories, shipments and customers. A review of these sections in the study guide is imperative.

How should I use this?

While the temptation might be to review each line of the source code provided and then go take the test. **NO, please do not do that.**

Here is the plan that will help you obtain the knowledge to pass:

- Download the source code and set up your development environment for this project.
- Enter the `app/code/SwiftOtter/OrderExport` directory and move all contents to another directory outside of `app/code`. For example, this would be the `reference` directory.
- Read through the Associate Developer study guide.
- Then, read through this project guide.
- Take the action suggested for each step.
- Review your work against my code.

Oh, and if you have found this helpful, I am pulling together videos for all of this material on my YouTube channel.

If you have suggestions or critiques, *I am all ears.* I want this to be an excellent and valuable resource for years to come. Please email me: joseph@swiftotter.com

Yours truly,
Joseph Maxwell

Source code

https://gitlab.com/swiftotter/associate-study-guide

YouTube Video Playlist:

https://www.youtube.com/watch?v=RlFTzkGZAz4&list=PLrW94TMW-eohnHkJuKzJm2RPhbqfMbDjd

Companion Study Guide/Practice Test:

https://swiftotter.com/technical/certifications/magento-2-certified-associate-study-guide

Requirements

- Create a module: `SwiftOtter_ExportOrders` in `app/code/SwiftOtter/ExportOrders`

- Focus is more on creating a module to fit the requirements of a test than to be a sellable module in the real-world. However, these requirements do come from a project we recently worked with. In this case, this merchant needed to review every order before pushing it off to their parent company's fulfillment center.

- Ability to specify a SKU override as a product attribute. This will allow us to use one SKU on the website, and another will be sent to the ERP.

- Use the https://github.com/boldcommerce/magento2-ordercomments module to save comments, but we need to send these comments in the result.

- Create a user interface on the view order page where a CSR can configure order details and push to the ERP:
  - Requested shipping date
  - Merchant notes field
  - Button to submit

- Only simple and virtual products are sent (including simple or virtual child products of a parent bundle, grouped or configurable)
  - SKU
  - QTY
  - Price per item
  - Cost per item

- Order header data

    - Password
    - Currency
    - Shipping name
    - Shipping address
    - Shipping city
    - Shipping state/province (2 digits)
    - Shipping postal code
    - Shipping country (2 characters)
    - Shipping amount
    - Shipping method (freight method overrides)
    - Requested shipping date

- Data will be pushed to an external web service.

JSON Output Template

```
{
  "password": 1234,
  "id": "1000001",
  "currency": "USD",
  "shipping": {
    "name": "Joseph Maxwell",
    "address": "123 Main Street",
    "city": "Kansas City",
    "state": "KS",
    "postcode": "12345",
    "country": "US",
    "amount": 15,
    "method": "UPS",
    "ship_on": "17/12/2020"
  },
  "customer_notes": "Please ship carefully.",
  "merchant_notes": "PO #123456",
  "items": [
    {
      "sku": "ASD123",
      "qty": 1,
      "item_price": 10,
      "item_cost": 4,
      "total": 10
    },
    {
      "sku": "QWE456",
      "qty": 2,
      "item_price": 12,
      "item_cost": 4,
      "total": 24
    }
  ],
  "discount": 0,
  "total": 49
}
```

Overview

- Data is POSTed to our Magento API via AJAX (a controller, for the purposes of this study guide as that is what is covered on the test, but ideally, an API).

- Split logic up into smaller files (instead of having one massive logic file).

Environment configuration

Video link

You will need to configure a local Magento 2 environment for yourself.

Here are some helpful places to start:

- MAMP
- Mark Shust's Docker System
- Valet+

Clone the source code:

```
git clone https://gitlab.com/swiftotter/associate-study-guide.git
```

(this will load the code into a new directory, `associate-study-guide`). Note that you might get a git error if you copy out the above code. If so, remove extra spaces around **`clone`**.

Note, if you intend to follow along (as you should), please take one of the following steps:

1. Move the existing `app/code` folder to something like `sample/`.
2. Delete the app/code folder in its entirety (and same the `composer.json` and `composer.lock` files).

The goal was to give you a fully-functioning module out-of-the-box. Then, you can edit as you see fit.

Jump into this directory:

```
cd associate-study-guide
```

and run:

```
composer install
```

Let Composer do its magic.

Install Magento

```
bin/magento setup:install \
    --admin-firstname="Magento" \
    --admin-lastname="Builder" \
    --admin-email="me@me.com" \
    --admin-user="admin" \
    --admin-password="HardPassword123" \
    --db-name="associate_developer" \
    --db-user="root" \
    --db-password="password"
```

Make sure to swap out all the values for these arguments. Remember to keep the quotes in place.

You will need to create a new MySQL database. For your development machine, feel free to use the `root` credentials (but of course, `root` credentials should **never** be used in staging or production).

Then, you will want to install sample data:

```
bin/magento sampledata:deploy
bin/magento setup:upgrade
```

Set up integration tests

First, copy `dev/tests/integration/etc/install-config-mysql.php.dist` to `dev/tests/integration/etc/install-config-mysql.php` (remove the `.dist` from the end).

Next, set the `db-host`, `db-user` and `db-password`. Remove the `amqp` array keys/values. Also, create the `magento_integration_tests` database.

In PHPStorm, go to Run > Edit Configurations. Expand the Templates level and click PHPUnit. Check “Defined in Configuration File” as well as the “Use alternative configuration file”. Set the value for the alternative configuration file to be `dev/tests/quick-integration/phpunit.xml`.

This `quick-integration` test environment comes from ReachDigital’s integration testing system. This makes running integration tests much faster. I have already included it in the git repository.

Now, you should be able to go to an integration test in `app/code/SwiftOtter/OrderExport/Test/Integration` and run that test by pressing `Ctrl+Shift+R`.

The first time you run it will take some time to execute.

Implementation details

The biggest question: "where do I start"?

Especially for beginners, I suggest starting with interaction points:

- Add UI elements (what we are doing here).
- Create Postman requests

In this case, we have already detailed what needs to happen and what is the final outcome. You need to write to make this happen.

Of course, I have already written this code. I suggest that you use my code as a double-check and not as a "copy and paste". This is meant to give you a guided experience and is not a substitute for your work and effort. I try to explain the method in getting to the end point.

Further reading:

- https://www.khanacademy.org/computing/computer-programming/programming/good-practices/a/planning-a-programming-project

Later, I highly recommend getting into writing unit and integration tests as the starting point. This has many advantages, the biggest of which is that your code is already tested. You know when you break it.

In this case, we are going to start with the form on the view order page. We will:

1. Initialize the module (Step #1).

2. Add layout XML instructions, create a `.phtml` template, set up a view model and link a `.js` file to power up the display on the View Order page (step #2).

3. Create the admin controller which will trigger the necessary processes (step #3).

4. Build out business logic to transform the order into a PHP array, which is easily converted to JSON (step #4).

5. Post the order to a web service (step #5).

6. Update the database to provide details as to when the order was exported (step #6).

7. Return the response to the frontend (step #7).

I hope you can see the chain of logic. Each step builds upon the previous step. Each step provides a stopping point at which you can look back and see the work you have completed.

Step 1: Initial configuration

Video link

Create SwiftOtter/ExportOrders extension skeleton (reference 1.1)

- app/code/SwiftOtter/OrderExport/registration.php
- app/code/SwiftOtter/OrderExport/composer.json
- app/code/SwiftOtter/OrderExport/etc/module.xml

Enable SwiftOtter/ExportOrders extension (reference 1.1, “What are the significant steps to add a new module?”)

```
bin/magento module:enable SwiftOtter_OrderExport
bin/magento setup:upgrade --keep-generated
```

Step 2: create the UI updates

Step 2.1: Identify which layout handle we need to customize.

(reference 2.2, “How do you get data from a controller to be rendered?”)

Video link

We get the layout handle from either the URL or Body Tag:

- URL: https://lc.associate.site/admin_dev/**sales/order/view**/order_id/3/key/.../ (note that this is my local environment’s URL and your’s may vary).
- Body tag: `sales-order-view page-layout-admin-2columns-left`

Please note that in the admin panel, this could be a uiComponent. In this instance, it is not, but this is a possibility for customizing forms and grids.

Step 2.2: Create switch to toggle functionality (global scope)

[Video Link]

This will happen in `etc/adminhtml/system.xml`. (reference 1.3, "Describe development in the context of website and store scopes.")

- How do we figure out what section and group to use?

  First, look through Store Configuration to find a relevant section. In this case, we will use the Sales (tab) > Sales (section). We will create our own group for this module.

- Now, search through the `vendor/magento` directory for all instances of Sales, matching case and filtering by file `*system.xml`. Since we know this is sales-related, I suggest looking for references in the `vendor/magento/module-sales` directory (or even starting your search there).

  You will find in (vendor/magento/module-sales/etc/adminhtml/system.xml): `vendor/magento/module-sales/etc/adminhtml/system.xml`

```
<section id="sales"
```

We have the ID of the section we want to create.

Create two System Configuration values:
- Enabled (with a Yes/no select list)
- API Password (ensure that the value is "encrypted", which is a keyword)

And here's how to do this:

- What parameters do we use?

  Go back to `vendor/magento/module-sales/etc/adminhtml/system.xml`, find a group, and copy its module value. Update them as applicable. Look for an `Enabledisable` or a `Yesno` source model. Copy that back into your system.xml file.

- We finally need (for the sake of demonstration) to have this automatically enabled. This happens in our module's `etc/config.xml` file.

- Clear the cache.

Now, for the sake of learning, have this module automatically enabled in `etc/config.xml`.

Step 2.3: Add a new `block` in `sales_order_view.xml`.

Video Link

(reference 3.5)

```
app/code/SwiftOtter/OrderExport/view/adminhtml/layout/sales_order_vie
w.xml
```

I suggest using the `ifconfig` layout XML attribute to determine whether or not to display this block.

Note: the best Magento user experience would be adding a button to the top order menu bar that triggers a slideout panel. However, for the sake of this example and the topics covered by the Associate Developer exam, we are using `.phtml` templates and layout XML.

How do I know what container I should reference for this block? Start by locating the original `sales_order_view.xml` file.

What is the original/base `sales_order_view.xml` file? This would likely be in the `Magento_Sales` module, but to be sure, let's do this exercise. Do a search for `frontName="sales` in `vendor/magento` with the File mask being `*routes.xml`. We know `sales` because it is the first entry in the URL *after* any admin references. Note that this search will turn up two `routes.xml` files: one in `frontend` and one in `adminhtml`. Either way, the `id` is the same. It is this module that contains the base `sales_order_view.xml` file:
`vendor/magento/module-sales/view/adminhtml/layout/sales_order_view.xml`

If you would so desire, you can enable block hinting: `bin/magento dev:template-hints:enable`. Or, you can look through the lists of containers and find one that roughly matches what you need.

In this module, let's go with the `content` container.

This block will be empty for now (remember that we do not need to specify the block class as that will be automatically determined to be Magento\Framework\View\Element\Template).

Step 2.4: Create a `.phtml` template

Video Link

```
app/code/SwiftOtter/OrderExport/view/adminhtml/templates/export.phtml
```

This will provide the option to set a shipping date and shipping notes. It also has a button to export the order.

Feel free to copy the `.phtml` file to your own project. Frankly, writing HTML is not overly pertinent to the test. I do suggest you take time to understand how the template interacts with the view model.

Step 2.5: Create a view model to provide business logic for this template.

[Video Link]

```
SwiftOtter\OrderExport\ViewModel\OrderDetails
app/code/SwiftOtter/OrderExport/ViewModel/OrderDetails.php
```

Remember that view models must implement `Magento\Framework\View\Element\Block\ArgumentInterface`.

The view model should have a `getConfig` method (that is called from the `.phtml` template) that returns an array with these keys:

- `sending_message`: the text of the submit button while the transmission is in progress.
- `original_message`: the original text of the button.
- `form_key`: this is a CSRF token that is retrieved in the `\Magento\Framework\Data\Form\FormKey` class. Never write this into a `.phtml` template that is cached in Varnish or the Full-Page Cache as this token will be frozen in time and your visitors will get redirects or error messages. We can write this `form_key` onto our `.phtml` template here because the admin area is excluded from full-page caching.
- `upload_url`: the URL to which we post the data and trigger the export. This will be created using the `Magento\Framework\UrlInterface` dependency. You might wonder why we didn't use the backend `UrlInterface`? That is because there is a preference in the `adminhtml` scope to map to the correct URL builder. You can return an empty parameter, like `$this->urlBuilder->getUrl()` for now. We will correct this when we create our route and controller.

It also needs two other methods:

- `isAllowed`: this is blank *for now* (until the next step).

- `getButtonMessage`: this returns the translated value for Javascript AND to be rendered in the PHTML (thus in a method). You wouldn't have to put this into a public method, BUT it allows this to be easily overridden through an after plugin by another module—this is up to you.

Step 2.6: Create an `etc/adminhtml/acl.xml` file and add a check to the view model.

[Video Link]

```
app/code/SwiftOtter/OrderExport/etc/adminhtml/acl.xml
```

All resources must be found inside a `Magento_Backend::admin` resource. Once we have our `acl.xml` properly configured (see https://www.mageplaza.com/magento-2-module-development/magento-2-acl-access-control-lists.html), we need to add a method into our view model. In the article, it shows an example of using the `_isAllowed` protected method in a controller. We can use this as a springboard to find out how this is implemented so we can use it ourselves.

Do a search for `protected function _isAllowed()` in `vendor/magento`. You will find a file like `Magento\AdminAnalytics\Controller\Adminhtml\Config\EnableAdminUsage`. Locate `_isAllowed` and `Cmd+Click` (Mac) or `Ctrl+Click` (Windows) on the `_authorization` reference. You will see this references `Magento\Framework\AuthorizationInterface`, so let's inject that into our view model. Put this into the previously-blank `isAllowed` method.

Now, clear the cache and visit an order page. You should see your content from the `.phtml` file now present.

Create the Javascript file to push details to exporter controller:

```
app/code/SwiftOtter/OrderExport/view/adminhtml/web/js/upload-form.js
```

Note: this should use uiComponents and Knockout templates (.html). I am reducing this example a little to provide a practical demonstration of what the test expects.

Feel free to copy this Javascript file into your project. Please review this file to understand what it is doing.

Now submit an order export. Of course it won't return information as the controller doesn't even exist yet. However, you just proved that you have triggered a successful HTTP request.

Step 3: create an admin controller

The first step in this process is to tell Magento that our module can listen to web requests (Web API is different).

Step 3.1: Create the `routes.xml` file

[Video Link]

(reference 2.2)

```
app/code/SwiftOtter/OrderExport/etc/adminhtml/routes.xml
```

Create an `etc/adminhtml/routes.xml file`. Use `order_export` for the ID and the `frontName`. In this case, the `router id` should be `admin`.

Step 3.2: Create the controller

[Video Link]

(reference 2.2)

```
SwiftOtter\ExportOrders\Controller\Adminhtml\Export\Run
app/code/SwiftOtter/OrderExport/Controller/Adminhtml/Export/Run.php

https://your.test.site/admin_dev/order_export/export/run (sample
domain)
```

How do we know what to put into a controller? Look for another admin controller in the Magento source code. We have been working with the `sales_order_view` controller, so let's go find that and use it as a loose template for our new controller.

To find it, we navigate to the `vendor/magento/module-sales` directory. Next, we look for the `Controller` directory. All admin controllers are then found in the `Adminhtml` directory. At this point, we have matched the first word in the handle: `sales_order_view` (sales). Now, we can look for the `Order` directory and the `View.php` file:

```
vendor/magento/module-sales/Controller/Adminhtml/Order/View.php
```

Alas, but this class extends another class in the `Magento_Sales` module. Let's navigate to that one. Here, we find the class we should extend in our controller: `Magento\Backend\App\Action`.

I also recommend having your controller implement the appropriate interface for the type of request this controller expects. In this case, it is:

```
Magento\Framework\App\Action\HttpPostActionInterface
```

Create the `execute` method. Because this method will return JSON, let's also go ahead and inject an instance of `Magento\Framework\Controller\Result\JsonFactory` into the constructor.

You can also create a test implementation that returns the JSON result to ensure that everything is working at this point.

Step 3.3: Determine the URL to trigger this route and add it to the view model.

Video Link

(reference 2.3)

```
// in app/code/SwiftOtter/OrderExport/ViewModel/OrderDetails.php
return [
    // ...
    'upload_url' =>
        $this->urlBuilder->getUrl('order_export/export/run')
];
```

The `order_export` comes from `etc/adminhtml/routes.xml`. The middle `export` comes from the `Export` directory in the path to the controller `(`app/code/SwiftOtter/OrderExport/Controller/Adminhtml/Export`) and the final `run` comes from the controller's class name.

There is one major piece of functionality that we are missing. **The order ID.** Let's add a parameter to this `getUrl` call:

```
$this->urlBuilder->getUrl(
    'order_export/export/run',
    [
        'order_id' => (int)$this->request->getParam('order_id')
    ]
)
```

By casting the value to an `integer`, we are reducing (maybe eliminating) the possibility for malicious Javascript code being written into the frontend.

We need to ensure that the view model depends on `Magento\Framework\App\RequestInterface` to populate its request property.

Step 3.4: Create the `HeaderData` model class

[Video Link]()

```
SwiftOtter\OrderExport\Model\HeaderData
app/code/SwiftOtter/OrderExport/Model/HeaderData.php
```

We need to create a class that represents a structured way to pass the ship date and merchant notes to our JSON transformer action. Of course, there is nothing wrong with two strings. I prefer a class just for this purpose as we can type-hint and type-check all in one. And, while two strings might be the original request from the merchant, I can *almost guarantee* that more fields will be coming. Setting up a class now will make your life easier when future updates come along.

It must have two fields:

- Ship Date (type DateTime)
- Merchant Notes (type string)

This model doesn't have to extend or inherit any class or interface. It will only have getters and setters.

Once you create the two **`private`** variables, you can set your pointer on one of the variables and press `Cmd+N` (Mac) or `Ctrl+N` (Windows) and then click Getters and Setters. This will automatically generate getter and setter methods for the desired properties.

Step 3.5: Add a `HeaderDataFactory` to the controller

[Video Link]()

Finally, we need to add this `HeaderData` class to our controller. The way to initialize this class is to use a factory:

```
SwiftOtter\OrderExport\Model\HeaderDataFactory
```

Remember that the Factory is automatically generated. When you add this class into your code for the first time, PHPStorm will say it doesn't exist—because it doesn't. As soon as PHP tries to autoload the file, Magento will create the file, and then PHPStorm will recognize it.

I suggest that you set a breakpoint in the controller's `execute` method. Then, click the "Send Order to Fulfillment" button on the frontend and let's see what happens!

We are now ready to build business logic for transforming order.

Step 4: Transform Order into PHP Array

Overview

[Video Link]()

We need to take data stored in the `sales_order` table and `sales_order_item` table and transform it into a PHP array (that will be converted to JSON in the next step).

Of course, we could do this in one class with many public methods. But my goal in this book is to demonstrate best practice. Ideally, we will have one public method per class with a number of supporting private methods. This reduces the number of dependencies per class making it easier to debug and test.

We will create a class structure like:

- Iterator action class (`SwiftOtter\OrderExport\Action\TransformOrderToArray`) iterates through child collector classes. The output is `array_merge`d together. This parent class represents a single point of entry. You can call this class anywhere, pass in the required parameters and you get an array response.

- Collector classes implement an interface (`SwiftOtter\OrderExport\Api\DataCollectorInterface`). Each one has a specific function. You could call any one of these anywhere in the Magento application if desired. Or, you can create a class that depends on this `DataCollectorInterface` and use a `di.xml` argument node to set *which class* is to be injected.

  These collector classes are associated with the iterator class in `etc/di.xml`.

  For example, look at `SwiftOtter\OrderExport\Collector\HeaderData`.

Step 4.1: Create data patch for product attribute

(reference 4.4)
[Video Link]

```
New product attribute: sku_override
```

Utilize the Magento CLI to create a data patch. If you don't remember the command to create the path, run `bin/magento` and look for the item having `patch` in its name (or, better yet `bin/magento | grep patch`).

```
bin/magento setup:db-declaration:generate-patch
SwiftOtter_OrderExport CreateSkuOverrideAttribute
```

This command results in:

```
SwiftOtter\OrderExport\Setup\Patch\Data\CreateSkuOverrideAttribute
```

There are quite a few ancillary comments in this file and I like to clean those out.

How do we know the syntax to create a new product attribute? Again, we turn to either the internet or Magento for an example. Magento is ideal. Let's go hunt for a new attribute.

As we think about it, the catalog module is responsible for products. There are many default attributes associated with products, so that would be the best first place to look.

Surprisingly, `Magento\Catalog\Setup\CategorySetup` creates both category attributes and product attributes, so it is not an easily digestible example.

Let's look through some other modules that would be related to products. The `module-bundle-product` module doesn't have any data patches that create an attribute. The `module-configurable-product` doesn't either.

We are striking out. The next thing is to think with some common sense: what might the method name be to add an attribute (we can supplement with the internet, too)? How about `addAttribute`? Let's do a search for `addAttribute(`.

We turn up this class: `Magento\GoogleShoppingAds\Setup\UpgradeData`. While, as of this writing, it has not been upgraded to a data patch, we can still learn from what happens here.

Start by injecting `Magento\Eav\Setup\EavSetupFactory` into the constructor. Now, copy the `addAttribute`-related code to our data patch.

When we call `$this->eavSetupFactory->create`, the `setup` parameter is found in `$this->moduleDataSetup` instead of a `$setup` variable.

Now, change the appropriate values (like the attribute code, name, and what product types it applies to, type and scope).

Add the `sku_override` attribute to `catalog_attributes.xml` (reference 6.7)

Further reading:

- [Magento 2: what is the catalog_attributes.xml file?]()

- How to access custom catalog attributes in Magento 2

Please note that you can `set used_in_product_listing` here when the attribute is created. Better yet, use the `catalog_attributes.xml` file. This file is somewhat equivalent, and the values can't be changed in the admin (so no mistakes will happen).

To create this file, do a search in your Magento source code for a file with the name `catalog_attributes.xml` (in PHPStorm, press `Shift` twice). Create a similar file in your module and add your attribute under the `catalog_product` `group`.

Run `bin/magento setup:upgrade --keep-generated`
You should now see the `sku_override` attribute in the `eav_attribute` table.

Step 4.2: Create the parent iterator class

Video Link

```
SwiftOtter\OrderExport\Action\TransformOrderToArray
app/code/SwiftOtter/OrderExport/Action/TransformOrderToArray.php
```

This class should have one **`public`** method, like:

```
public function execute(
    int $orderId,
    SwiftOtter\ExportOrders\Api\Data\HeaderData $headerData
) {
    // code goes here.
}
```

This method clearly represents the entry point to this class. As an action, other developers immediately identify this file that it does something.

Remember, there is nothing magical about this class or its naming. We are simply creating a self-contained unit of functionality. Any other area of the application could, in theory, easily utilize this code *provided* it passes in the correct data requirements.

As you will see in the above `execute` method, we take an Order ID and a `HeaderData` class. We now need to load up the order. We could use a repository or collection. Since an order is not an EAV-enabled entity, always use the repository. In PHPStorm, I add a new line to my constructor and start typing `OrderRe` and PHPStorm autofills: `Magento\Sales\Api\OrderRepositoryInterface`. Please make sure you have utilized the `Magento\Sales` order repository as there is the chance that other module developers will create a class with the same name.

In our `execute` method, we load the order and then iterate through each transformer. If we type-hinted our `$transformers` class property with the interface (in my case, `SwiftOtter\OrderExport\Api\DataCollectorInterface`), we will get code completion.

Before we continue, let's go ahead and create an entry in `di.xml` that will eventually contain our list of collectors:

```
<!-- app/code/SwiftOtter/OrderExport/etc/di.xml -->
<?xml version="1.0" ?>
<config xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"

xsi:noNamespaceSchemaLocation="urn:magento:framework:ObjectManager/et
c/config.xsd">
    <type name="SwiftOtter\OrderExport\Action\TransformOrderToArray">
        <arguments>
            <argument name="collectors" xsi:type="array">
                <!-- collectors go here -->
                <!-- sample: -->
                <!--
                    <item
                      name="header_data"
xsi:type="object">SwiftOtter\OrderExport\Collector\HeaderData</item>
-->
```

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

Step 4.3: Create an orchestrator to bring all the pieces together

Video Link

```
app/code/SwiftOtter/OrderExport/Orchestrator.php
SwiftOtter\OrderExport\Orchestrator
```

We need to connect this action up to the controller… somehow.

Of course, we could add it as a dependency to the controller. That would work fine, except for the fact that we have some additional steps in this business logic process.

I propose we create an `Orchestrator` that will execute all tasks to:

- Generator order JSON
- Push to external web service
- Update applicable entries in the database
- Return some type of result

This will become the central entry point to complete this functionality.

As such, I just created `SwiftOtter\OrderExport\Orchestrator`. We will create a `results` method that will return an array.

> *Note: if we were to be utilizing the WebAPI, you would want the orchestrator to return a "response" class (with a getter/setter for things like "success" and "error message"). This class would implement an interface located in the Api/Data directory of your module.*

Now, we need to inject this class into our controller (`SwiftOtter\OrderExport\Controller\Adminhtml\Export\Run`).

Feel free to click our now-famous "Send Order to Fulfillment" button and see how this works.

Step 4.4: Create the child data collector interface

Video Link

```
app/code/SwiftOtter/OrderExport/Api/DataCollectorInterface.php
SwiftOtter\OrderExport\Api\DataCollectorInterface
```

We have created our parent iterator and we have child collectors.

Let's take a first cut at what our collectors will look like. Remember, each collector will have a specific subset of order data to collect and return to our parent iterator.

In this case, I've given you a head start because I have worked through what inputs are necessary.

Here is how I often create interfaces:

- Create the interface
- Create a class (implementing this interface)
- Establish the method signature in the class
- Copy that back to the interface

In my experience, interfaces mature. They change and morph. As long as it is just you, that's not a problem. If you work with other developers, you need to communicate. If you work on Magento, you just don't change interfaces (I'd put an emoticon for laughing here).

In this case, a collector should `collect`. One `public` method. Of course, we can have other `private` methods.

Let's think through what parameters are needed. First, each collector needs an order. We could get specific and say that the Simple product collector should be sent an array of the order items. Or, better yet, we let the Simple product collector load the order items. The possibility for redundancy isn't a problem as:

- order items are fast to load (unlike EAV)
- this type of operation, executed by an admin, doesn't have to be as performant as the frontend.

We also have the header data collector that loads the shipping address and totals. For the sake of standardization, we could pass this class to each collector.

Here is our method declaration:

```
public function collect(OrderInterface $order, HeaderData
$headerData): array;
```

Step 4.5: Create header data collector interface

[Video Link]

```
app/code/SwiftOtter/OrderExport/Collector/HeaderData.php
SwiftOtter\OrderExport\Collector\HeaderData
```

Our first of several data collectors will be the one to gather up the "header" data or anything that is not item data.

This will specify the following fields:

```
{
  "password": 1234,
  "id": "1000001",
  "currency": "USD",
  "shipping": {
    "name": "Joseph Maxwell",
    "address": "123 Main Street",
    "city": "Kansas City",
    "state": "KS",
    "postcode": "12345",
    "country": "US",
    "amount": 15,
    "method": "UPS",
    "ship_on": "17/12/2020"
  },
  "customer_notes": "Please ship carefully.",
  "merchant_notes": "PO #123456",
  "discount": 0,
  "total": 49
}
```

Let's get to work.

Configure the password field.

Add a new `field` into the `etc/adminhtml/system.xml` file.

You can search for something like "magento 2 password system.xml". However, this is a good example of only one reference giving correct information (and there are many search results):
https://mage2.pro/t/topic/44

This is where a senior developer or a tech lead whom you can ask questions of is **invaluable.**

The challenge here is that searching the vendor/magento directory for all references to "password" (in `*.system.xml` files) would not yield the correct approach, either. This is a fairly rare occurrence and will be solved through additional experience. You are reading this and shouldn't have to work through this challenge again.

Get the shipping address

I am putting this in its own section before "Build the header data array" so that we can delve into a solid way to get the address.

We need the shipping address so we can send back the ship-to name, address, etc. The first place to look would be on the `OrderInterface`. In our `execute` method, let's use autocomplete to help:

```
$order->getAddress
```

We only see references to: `$order->getBillingAddressId()`. Nothing for a shipping address.

We could bypass the `OrderInterface` and call `getShippingAddress` directly from the `Order` model implementation. But this would not be future-proof as Magento may remove that method.

When we are working for a merchant, we have to balance the clock against the best implementation (hopefully we do have enough time to do our best). However, when we are preparing to pass a test, we have no clock to watch, so we must go above and beyond.

Let's do this another way: by using the order address repository. In your constructor, start typing `OrderAddress` and select `Magento\Sales\Api\OrderAddressRepositoryInterface`. If we are needing to filter results, we will also need to inject a `SearchCriteriaBuilder`.

Next, we can create a new method to `getShippingAddressFor(OrderInterface $order)`. In here, we need to find the correct address:

- Open up your SQL browser and find the `sales_order_address` table.

- Locate the columns that store the order ID (`parent_id`) and the address type (`address_type`). If you need help finding the order ID column, look at the foreign key relationships to find which one maps a column in this table to the `sales_order` table.
- Use the `SearchCriteriaBuilder` to add two filters, then `create()` it.
- Remember, if the product ordered was virtual, there may be no shipping address, so make sure you accommodate a null result.

I suggest using autocomplete to help out here. Start typing `$order->ship`. This in itself will not yield a correct method. Yet, the autocomplete should guide us to better answer. We *could* call the `$order->getShippingAddress()` method on the `Order` implementation of `OrderInterface`. However, we are in this for our personal development, so let's take the opportunity to load up the shipping address in another way.

Watch out for the address. It comes back as a `string[]` type.

Build the header data array

- `password`: note that we pass the `scopeType` argument as `SCOPE_STORES` and set the store ID. When this code is executed, we will be in the admin store scope. Because the password can be changed in global, website or store view scopes, we must obtain the correct value. This will load the store-specific value, and will fall back to the website then global scopes as needed.

- `currency`: the order's `base` or `order` currency code can be obtained. Remember, the `base` currency code is the code for the website (or global scope, if the Product Price Scope = Global). The `order` currency code is the code in which the order was placed (from the list of allowed currencies in Stores > Configuration > General > Currency Setup).

- `shipping`: see discussion above. This is optional—if an order only has virtual products, then there is no shipping address.

- `customer_notes`: this comes from the Bold Commerce customer notes module. While we could look in that module's code, the first place I like to look at is in extension attributes: `$order->getExtensionAttributes()`

  Browse that list and see if something matches. It does: `getBoldOrderComment()`.

- `merchant_notes`: this will come from the `HeaderData` input.

- Totals: in this case, I suggest working with the `getBase...` values. This means the values that are transported to the ERP will be the original currency values, and not those that are converted.

Finally, we must add this `HeaderData` collector to the list of collectors, using `di.xml`.

Let's see how it works now!

Note: you can run `SwiftOtter\OrderExport\Test\Integration\Collector\HeaderDataTest::testHeaderOutputMatches` to see if your code works as expected. If the test passes, we can move on to the next step.

Step 4.6: Create item data

[Video Link]

We are now ready to convert each order item into an array.

Create the item collector class

```
app/code/SwiftOtter/OrderExport/Collector/ItemData.php
SwiftOtter\OrderExport\Collector\ItemData
```

Iterate through all order items

We need to assemble this list of items and return it to the master array.

How do we get a list of these items? First, turn to the `OrderInterface` object. Are there any methods that have to do with `getItems`? It turns out that our answer is "yes". If we navigate to the `Order` class (see here), we will find that the `getItems` method will initialize itself if no items exist.

That was easy.

We forgot something that is **very** important. We need to ensure that only the correct product types (simple and virtual) are exported.

Let's use Magento's powerful dependency injection framework to resolve this:

- Add an `array $allowedTypes` parameter to the `__construct` method.
- Add the configuration to `di.xml`:

```
<type name="SwiftOtter\OrderExport\Collector\ItemData">
        <arguments>
            <argument name="allowedTypes" xsi:type="array">
                <item name="simple"
xsi:type="const">Magento\Catalog\Model\Product\Type::DEFAULT_TYPE</it
em>
                <item name="virtual"
xsi:type="const">Magento\Catalog\Model\Product\Type::TYPE_VIRTUAL</it
em>
            </argument>
        </arguments>
    </type>
```

Why would we do it this way? Couldn't we use a constant in our class? Yes, of course. But this way provides a good deal of flexibility:

- Other developers can easily extend.
- Our unit tests can be flexible.
- We can have different values for the frontend and adminhtml scopes.

If you don't need any of the above, there is no problem with putting this as a constant in the class. Remember, we are working to hone our "developer" capabilities and going above and beyond is the #1 way to do this.

Note: if you are using integration tests and find that you add this code, but then run the test and get an error like:

```
BadMethodCallException : Missing required argument $allowedTypes of
SwiftOtter\OrderExport\Collector\ItemData.
```

That is because the integration test environment's cache needs to be cleared. Delete the folder `dev/tests/integration/tmp/sandbox-[numbers here]/var/cache`.

Transform each item

Let's create a new `private` method that will transform the items as they are passed through. Of course, we could do this in the `foreach` in the previous section. But, we should work to create methods that are 15 lines or less in length. That makes it easier to read.

Here is what we need to end up with:

```
{
  "sku": "ASD123",
  "qty": 1,
  "item_price": 10,
  "item_cost": 4,
  "total": 10
},
```

This is the easy part.

Add the `ItemData` collector to the list of collectors in `di.xml`.

Run the integration test

```
app/code/SwiftOtter/OrderExport/Test/Integration/Collector/ItemDataTe
st.php
SwiftOtter\OrderExport\Test\Integration\Collector\ItemDataTest
```

This completes step 4.

Step 5: Post the order to a web service

[Video Link]

This step is the hardest to fully implement because we don't have an easy-to-access webservice to validate.

Create action to push order details

```
app/code/SwiftOtter/OrderExport/Action/PushDetailsToWebservice.php
SwiftOtter\OrderExport\Action\PushDetailsToWebservice
```

For purposes of this course, this file is quite empty. However, if you would like to push to an external resource, this is where you would do it.

We return **`true`** at the top of the file so that the process will continue to execute. Feel free to substitute this out for something else.

Before we close off this section, you might also notice that the return type for `execute` is `bool`. A **`true`** or **`false`** returns very little information about errors that occur. We could change the return type out to a `string`. But that also doesn't work very well, as how do we know that this action successfully completed? A **`true`** is very clear that this worked well.

Instead, throwing exceptions are very effective. You can catch them to learn what happened in the method. Left uncaught, they will stop the process of the application, which might not be a bad thing. This is all under your control.

When you are having trouble figuring out how to alert calling methods of a problem in a method, use an exception.

Finally, we need to add this `PushDetailsToWebservice` to our `Orchestrator` class.

Step 6: Update the database to provide details as to when the order was exported

In this step, we save details to the database about the order export process.

Overview

Video Link

There are quite a few steps to this one, so please bear with me.

First, we need to establish the data structure for these fields:

- Requested shipping date (from the ship date on the view order page)
- Exported at
- Merchant notes (from the merchant notes on the view order page)

Where should these values be stored in the database?

When I was learning Magento, this was a difficult question for me to answer. Since they are attached to an order, what about adding columns to the `sales_order` table? But, there could be a possibility that we would want multiple rows per order in the future. Creating a new table can seem like a big task, but thanks to `db_schema.xml`, it is much easier than you would think.

Step 6.1: Create `db_schema.xml`

Video Link

```
app/code/SwiftOtter/OrderExport/etc/db_schema.xml
```

We can start by populating this file's skeleton with our PHPStorm template.

Note: you can utilize some XML auto-completion with this command (for PHPStorm projects):

```
bin/magento dev:urn-catalog:generate .idea/misc.xml
```

You can also look up DevDocs articles about `db_schema.xml`.

- `id`: is an identity. Note that you also need a primary constraint to make this column a primary key.

- `order_id`: this should match the value specified in `vendor/magento/module-sales/etc/db_schema.xml` for the `entity_id` column EXCEPT for the `identity="true"` clause. Because this matches, we will also be able to create a foreign key to link the `entity_id` in the `sales_order` table to the `order_id` in our `sales_order_export` table.

- `ship_on`: `date` type. We don't need the time as that is not available in the selector in the admin panel.

- `exported_at`: `datetime` type. Here, the time would be helpful. Remember to store the time in UTC. This is ideal as we can convert it from the standard timezone (UTC).

- `merchant_notes`: `text` type.

Now run `bin/magento setup:upgrade --keep-generated`. You should see `sales_order_export` in your list of tables.

Step 6.2: Generate the model

Video Link

```
SwiftOtter\OrderExport\Model\OrderExportDetails
```

```
app/code/SwiftOtter/OrderExport/Model/OrderExportDetails.php

Extends: Magento\Framework\Model\AbstractModel
```

How do I know which class to extend?

For each member of the model triad (model, resource model and collection), there is a Magento class that we should extend. The challenge is remembering which class that is.

If I ever forget, I often turn to the Magento CMS module. The Block model is a good and basic pattern for your classes.

Configure parameters

You need to override the `_construct()` method (one underscore). Here, you will assign the class path to your resource model.

This file has not yet been created, but we can anticipate the page:

```
$this->_init(\SwiftOtter\OrderExport\Model\ResourceModel\OrderExportD
etails::class);
```

By convention:

- The resource model has the same class name as the model, it is in the `Model/ResourceModel/` directory.
- Never hardcode class paths as a string. Use the `::class` constant. This allows you to shorten paths using aliases, keeping your code cleaner.

Add getters and setters

If we extend the `AbstractModel` class, which we should, data will be loaded into this model's `$data` array.

We access that array with the `getData` and `setData` methods.

I like to use getters and setters for a couple of reasons:

- Relying on the magic methods that Magento brings can be tricky for a newbie in the system to understand.
- Code completion does not work for magic methods.
- Types can be enforced.
- Plugins work on public methods so other modules can adjust values from a model.

Do I need an interface for this model?

If you plan to expose this model via the API, then “yes”. Otherwise, don’t worry about it.

For the sake of our learning today, we will go ahead with creating the interface.

```
app/code/SwiftOtter/OrderExport/Api/Data/OrderExportDetailsInterface.
php
SwiftOtter\OrderExport\Api\Data\OrderExportDetailsInterface
```

The fastest and most accurate way that I have found to create an interface from a class is:

- Copy the methods from the model.
- Paste them into the interface.
- Delete out everything between the `{}` around a method and replace it with a `;`.

Step 6.3: Generate the resource model and collection

[Video Link]

These two files are relatively simple to create.

Resource model

```
app/code/SwiftOtter/OrderExport/Model/ResourceModel/OrderExportDetail
s.php
SwiftOtter\OrderExport\Model\ResourceModel\OrderExportDetails
```

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

This file bears the same name as the model—with the only difference being that the resource model is stored in the module's `Model/ResourceModel` directory (by convention).

We need to implement the `_construct` method and call the `_init` method to specify the table in the database and the column that is the row identifier (primary key).

Collection

```
app/code/SwiftOtter/OrderExport/Model/ResourceModel/OrderExportDetail
s/Collection.php
SwiftOtter\OrderExport\Model\ResourceModel\OrderExportDetails\Collect
ion

Extends:
\Magento\Framework\Model\ResourceModel\Db\Collection\AbstractCollecti
on
```

Notice that this file is almost always named "Collection", but it is inside the `Model/ResourceModel/[model name]` directory.

The `_construct` method (again, one underscore) should call the `_init` method with the model class and the resource model class, respectively.

Note: I use PHP import aliases quite often. My standard protocol is to put the name of the class first, then the last "directory" in the namespace path second.

For example, let's alias the OrderDetails model:

```
SwiftOtter\OrderExport\Model\OrderExportDetails
```

`OrderExportDetails` is the model name. `Model` would be the last "directory" in the namespace path:

```
use SwiftOtter\OrderExport\Model\OrderExportDetails as
OrderExportDetailsModel;
```

That is very readable.

This works especially well if we need to import two classes with the same name (but, of course, different namespace paths).

At this point, we don't have much new tech that can run. But we will get there shortly.

Step 6.4: Create a repository

Video Link

I'll be frank, a repository is not 100% necessary for this customization. However, this is helpful in that you gain additional practice in learning your way around the Magento system: all with the great hope of soon passing the Associate developer exam.

In many ways, a repository is the centralized gateway to a model, resource model, and collection.

Create the file

We will use the `vendor/magento/module-cms/Model/BlockRepository.php` file as a template for our details repository:

```
SwiftOtter\OrderExport\Model\OrderExportDetailsRepository
app/code/SwiftOtter/OrderExport/Model/OrderExportDetailsRepository.ph
p
```

Build the constructor

Let's review each constructor parameter in the `BlockRepository` class and see if we need to utilize this on ours:

- `$resource`: YES. This is the resource model that we just created. This will be used to save and delete rows in the database.
- `$blockFactory`: YES (but change the name). This is how we will create a new instance of our `OrderExportDetails` model.
- `$dataBlockFactory`: NO. We do not need this because we do not separate out the block from its data layer.
- `$blockCollectionFactory`: YES (but change the name). Remember that collections store their filters and results. As such, we always want a new clean collection so we need to use a factory.
- `$searchResultsFactory`: YES. We will need to create a search results interface for this repository. Follow the example set in `Magento\Cms\Api\Data\BlockSearchResultsInterface`. Don't forget to search for `BlockSearchResultsInterface` in the `vendor/magento/module-cms`, filtered by `di.xml` to find any references to this class (hint: there is a preference).
- `$dataObjectHelper`: NO. The block repository itself no longer uses this.
- `$dataObjectProcessor`: NO. The block repository itself no longer uses this.
- `$storeManager`: NO. We do not need to know the current store ID.
- `$collectionProcessor`: YES. This is used by the `getList` method to convert a SearchCriteriaInterface to methods utilized in a collection.

Ensure dependency injection is configured. The easiest way to replicate what Magento has done for the block repository is to search for `BlockRepository` in the `vendor/magento` directory, filtered by files that match `*di.xml`.

We find two: the first is a preference for the `BlockRepositoryInterface`. Since we are not exposing this repository to the Web API, we do not need a service contract (interface) for our order export details repository. The second is a series of virtual types to generate the collection processor.

We will copy, then modify, the `<type name="Magento\Cms\Model\BlockRepository">` and the `<virtualType name="Magento\Cms\Model\Api\SearchCriteria\BlockCollectionProcessor"` nodes.

- Change the names for the `virtualType` and the `type` (the latter should match the repository class name). Make sure to update the `argument type` to match the new `virtualType` name.
- I saw that the `sorting` and `pagination` values for the processors were similar, but the `filters` processor was in the CMS module. I checked the the directory that holds the `sorting` class and found that there is a `FilterProcessor` there—so we can use that.

Create methods

We can use the `BlockRepository`, again, as a template for how we should build the `OrderExportDetails` repository.

- `save`: this is a centralized place to save a model (that we create). Remember that there is nothing magical about a repository. There is also no central implementation and all methods created here are by convention.

- `getById`: this loads a model by it's ID. Please do not use the `load` method that is still available on a model. That is deprecated and will go away at some point in the future.

- `getList`: this is like a collection. Arguably it's a little more convenient as we don't have to create a factory for this repository.

- `delete`: this deletes a model (through the resource model).

- `deleteById`: this loads a model and then deletes it.

Create the `SearchResultsInterface`

`SwiftOtter\OrderExport\Api\Data\OrderExportDetailsInterface` `app/code/SwiftOtter/OrderExport/Api/Data/OrderExportDetailsInterface.php`

This final step is not 100% necessary, but you can use docblock type hints (not PHP type hints) for the `getList` method. This informs the **`foreach`** and other loops what type of object to expect.

Oh, and don't forget to add a preference to di.xml:

```
<preference
for="SwiftOtter\OrderExport\Api\Data\OrderExportDetailsSearchResultsInterface"
    type="Magento\Framework\Api\SearchResults" />
```

This tells Magento that should the `OrderExportDetailsSearchResultsInterface` ever be called, to return the `SearchResults` class.

Step 6.5: Create extension attributes

[Video Link]

Thus far, Step 6 has been boring. We are about to change this.

We are now going to create extension attributes!

```
app/code/SwiftOtter/OrderExport/etc/extension_attributes.xml
```

Please take a moment to review in the study guide and on Magento DevDocs what is an extension attribute.

In our case, we will map the `code="export_details"` to the `OrderExportDetailsInterface` that we just created.

Once you have declared your extension attributes, please delete the `generated/Magento/Sales` directory. This will force the generated extension attribute interfaces and classes to be regenerated.

Loading the data

The interesting part of extension attributes is that they are a shell. They do nothing by themselves. We make them happen in their entirety.

Since we are attaching these extension attributes to an `OrderInterface`, we have several places that we need to make this attachment happen:

- OrderRepository::get
- OrderRepository::getList
- You could also attach to the Order collection as well, but we will not for this example.
- Finally, another place for a plugin would be the `OrderFactory`'s `create` method. You can create plugins for generated classes (in addition to plugins for other plugins). By doing this, you would guarantee that `getExportDetails` would always return an object.

We will create a plugin for the `OrderRepositoryInterface`:

```
app/code/SwiftOtter/OrderExport/Plugin/LoadExportDetailsIntoOrder.php
SwiftOtter\OrderExport\Plugin\LoadExportDetailsIntoOrder
```

We will create two methods:

- `afterGet`. The first parameter is the class that we are modifying. Ideally, this will match whatever we use as the `type name` property in `di.xml` that will house our `plugin` node. The second parameter is the result of the original method call. In this case, based on the docblock, it would be `Magento\Sales\Api\Data\OrderInterface`.

- `afterGetList`. The second parameter's type is also seen in the docblock: `Magento\Sales\Api\Data\OrderSearchResultInterface`. This method will loop through each order and call a method to set the extension attributes.

Don't forget to return the value from the `after` methods (for that matter, `before` or `around` plugins, too).

And, please remember to add the `type` and `plugin` nodes to the module's `etc/di.xml` file.

Initializing extension attributes

Extension attributes are interesting in that the result returned by calling `getExtensionAttributes` on an `Order` model, for example, may be **`null`** or a generated implementation of `Magento\Sales\Api\Data\OrderExtensionInterface`. You are responsible for ensuring that the result is an object. So, every time you need to set your extension attribute value, the class must depend on an instance of the factory that will create the parent extension attribute class.

In addition, there is no automated way to set your specific extension attribute's value. You must do that.

The plugin class described above will handle these details.

How do I know which extension attribute factory to inject? This takes a little work as the parent extension attribute class (what is returned from calling `getExtensionAttributes` on a extension-attribute-enabled class) is generated.

For example, we are needing to set extension attributes on the `Magento\Sales\Api\Data\OrderInterface`. The extension attribute factory would be:

```
Magento\Sales\Api\Data\OrderExtensionInterfaceFactory
```

Note the addition of `Extension` and `Factory`. `Extension` is always added before **`Interface`** and `Factory` just after **`Interface`**.

Here is another example: `Magento\Catalog\Api\Data\ProductInterface`

Would yield:

```
Magento\Catalog\Api\Data\ProductExtensionInterfaceFactory
```

Feel free to set breakpoints in our `LoadExportDetailsIntoOrder` class and visit the order page.

At this point, I hope you take a moment to stand back and admire how good the code is looking.

Step 6.6: Save the export details to extension attribute

Video Link

```
app/code/SwiftOtter/OrderExport/Action/SaveExportDetailsToOrder.php
SwiftOtter\OrderExport\Action\SaveExportDetailsToOrder
```

Overview

This final class that we must create will:

- Set values on the extension attribute.
- Save the extension attribute (which is really a model, so we can use the repository to do this).

Set values on the extension attribute.

This is all pretty straightforward. We are mapping the merchant notes and ship on date to the applicable values in the details object. If the export was successful, we set the export date. Then we save the export detail model.

Don’t forget to also set the order ID on the extension attribute.

Step 7: Return the response to the frontend

Video Link

The final thing we need to do is ensure that the response gets back to the frontend. This happens by returning the result of `Magento\Framework\Controller\Result\JsonFactory::create`.

Again, we are just using a controller for the sake of learning about controllers. The far better route is to use a Web API request. The way we have configured this module means a very simple conversion to use the WebAPI.

Indicate on the admin order view that this order has been exported.

For good user experience, it would be ideal to indicate to the admin user that this order has been exported.

To do this, we need to complete several steps:

- Create a service provider. This will expose the order for our consumption. Of course, the view model could depend on a `RequestInterface`. However, by centralizing it, we have several advantages: 1) we can mock results with our tests and 2) if we need custom logic around loading the order, we don't have to duplicate that across multiple classes.
- Create a new view model to provide business logic to a new template.
- Create a new template to render results on the frontend.
- Use a layout XML `block` node to render the template (and associate the view model).

Step 7.1: Create a service provider

Video Link

```
app/code/SwiftOtter/OrderExport/Service/Order.php
SwiftOtter\OrderExport\Service\Order
```

This class should be very familiar as we use concepts that you should already know. We take the `order_id` value from the `RequestInterface` and use that to load up the order from the `OrderRepository`.

Note several things:

- We do not need to worry about the overhead of loading an object multiple times for several types of entities including at least orders and customers. That is because there is an internal registry that caches these objects once they are loaded.
- The order repository's `get` method will check to ensure that a value is sent, so we do not need to send that.
- If possible, I like to cast a `string` to an `int` as this acts as a sanitizer. This is only possible in certain situations like the ID of a particular model.

Step 7.2: Create a new view model

[Video Link]

```
app/code/SwiftOtter/OrderExport/ViewModel/HasBeenExported.php
SwiftOtter\OrderExport\ViewModel\HasBeenExported
```

We could reuse the same view model that we created earlier (`SwiftOtter\OrderExport\ViewModel\OrderDetails`). However, one goal that I have with this guide is to overcome your challenge of creating new files. At least, I have had that. I wouldn't even call it a phobia: it's just that I didn't like to create a new file. As a result, I have had no fallout or regrets of more classes.

Step 7.3: Create a new template

[Video Link]

```
app/code/SwiftOtter/OrderExport/view/adminhtml/templates/status.phtml
```

This template needs to do nothing special: its only purpose is to show the status of whether or not the order has been exported.

I am trying to demonstrate that it is very easy to use multiple templates. There is no need to stuff everything into one template.

Step 7.4: Render the template with Layout XML

[Video Link]

```
app/code/SwiftOtter/OrderExport/view/adminhtml/layout/sales_order_vie
w.xml
```

We are creating a block (no block type specified as it infers the `Template` type) with the view model argument. This should be EASY by now.

Step 8 (optional): utilize the WebAPI

So far, our work has been to follow the objectives of the Associate Developer Study Guide. There is one big feature (one of my favorites) that has not been covered: the **WebAPI.** The API is simple to use and works very well.

This section will focus on what it takes to replace the controller with a WebAPI request. We will only need one WebAPI route, as this is replacing the controller.

Overview

Video Link
To create a WebAPI request, we must build three components:

- Incoming data structure (scalar and object values)
- Business logic
- Response data structure (scalar OR an object)

(and, of course, add configurations to etc/webapi.xml)

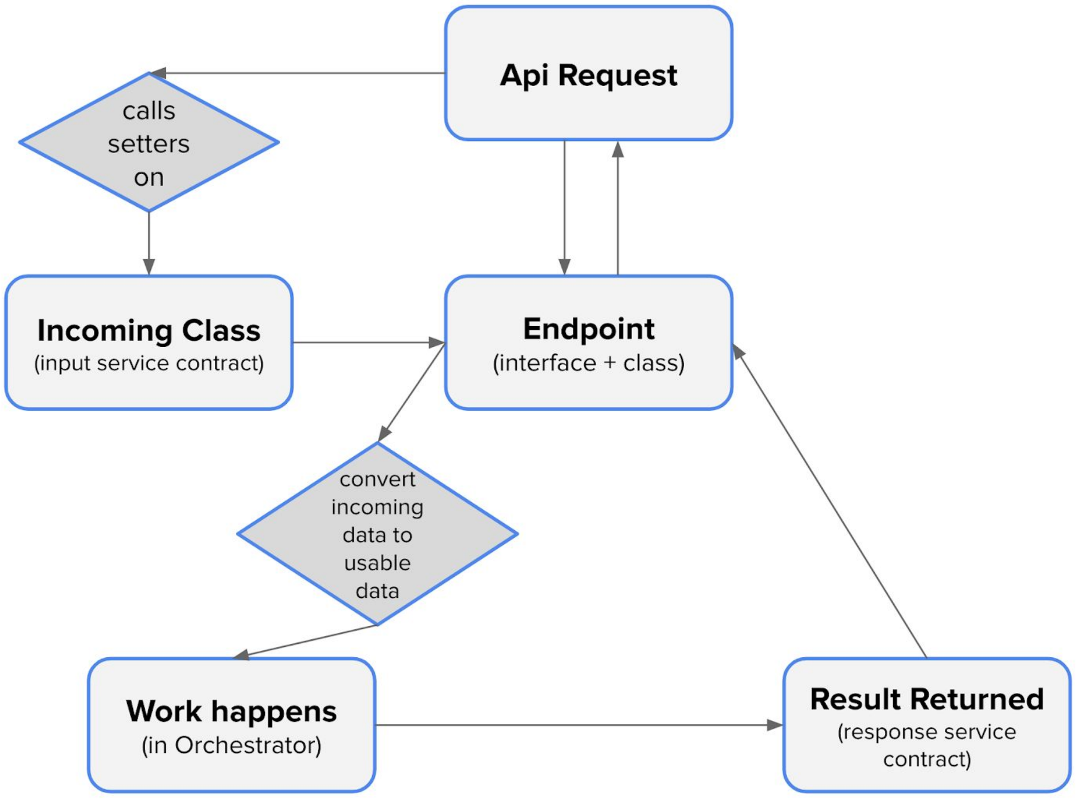
We will also need to create an endpoint service contract (interface) and class. The method will be type-hinted with a new class: one that wraps the Header Data class. While we could convert all `DateTime` types in `HeaderData` to be strings, that would make it incompatible with what we have already built—and we miss a great opportunity to dive into something more.

Of course, we also need to create return types for this information going back to the browser.

You will see that this implementation is similar in concept to the CMS' implementation of the API: `vendor/magento/module-cms/etc/webapi.xml`. The big differences might seem to be:

- We are not using a repository. Yes, but remember, this is just a specified service contract. There is nothing magical in the name. All that is used is an interface (with a preference associating this with a class).
- We are not using a PageInterface as a parameter. We are, though, using an interface. Again, the concept is illustrated below, data is sent in, and a result is returned. Please take a moment to look through `vendor/magento/module-cms/Api/PageRepositoryInterface.php` and you will see this point. For example, the save method takes a `PageInterface` and returns a `PageInterface`. In our code, we are taking a set of input data and returning a response object. Concept is identical, but the details are quite different.

It took me a while to wrap my mind around this, so please do not stress if this is also a challenge for you.



Step 8.1: Build out the etc/webapi.xml file.

Video Link

```
app/code/SwiftOtter/OrderExport/etc/webapi.xml
```

The first step is that we need to define the route and entry point for our API request. That happens in `webapi.xml`.

We start by looking for a template with which we can copy. A search through the `vendor/magento` directory will yield plenty of results (note that I have to check "Include non-project results" to get listings to appear from the `vendor/magento` directory).

There are many options here, but I like to stick with examples from the CMS module (`vendor/magento/module-cms`). These are usually simple, instead of the complexities that are in other modules, like the Catalog module.

Define the route URL

You can give this any path you would like to give it. However, most routes start with `/V1/`, presumably so you could introduce a `/V2/` in the future—and have two separate routes that both exist at the same time to allow for breaking changes and an upgrade path.

In this route URL, you can also add a parameter with this notation: `:orderId`. This will match up to an argument in the method that is eventually called.

I recommend this url value:

```
/V1/order/export/:orderId
```

The URL to access this request will be:

```
[your local site name]/rest/[whatever you entered]/[order id]
```

For example, for my machine, with my recommendation, it is:

```
https://lc.associate.site/rest/V1/order/export/[order id]
```

Define the correct method

If you don't, you will get a invalid route error—and then you will try to troubleshoot it only to come back and realize that you entered the wrong route in.

Set this to be POST.

Define a service

The service will be an interface that we create (which will have a class associated through a `di.xml` preference). We can go ahead an pre-specify this:

```
SwiftOtter\OrderExport\Api\ExportInterface
```

The method will be:

```
execute
```

Note that these "endpoint" classes do not have to follow our "one public method per class". However, there is nothing preventing this methodology.

Define the resource

We need to specify the ACL resources for this. Feel free to review the other `webapi.xml` files in the project (or DevDocs) to understand other configuration options.

Remember our `etc/adminhtml/acl.xml` file that we created in 2.6? Open that up and grab the resource id (`SwiftOtter_OrderExport::OrderExport`) and create our resource.

However, for the sake of easy testing, let's do something different, temporarily. Let's set the resource to be this:

```
<resource ref="anonymous" />
```

By also temporarily switching the method to GET, we will soon be able to use the web browser to navigate to our API endpoint.

Comment the previous like with the specified ACL reference and leave our `anonymous` resource uncommented. We will fix this later.

Don't forget to clear the cache once you make changes to this file or any interface!

Step 8.2: Build the endpoint service contract (interface) and related data interfaces

Video Link

```
app/code/SwiftOtter/OrderExport/Api/ExportInterface.php
SwiftOtter\OrderExport\Api\ExportInterface
```

This is the contract that will accept requests from this API call.

We will add the `execute` method. There is nothing magical about this method's name, but it clearly delineates its purpose. Oh, and once you have an execute method, it's hard to think of names for other public methods in that same class.

What parameters will we specify? Please remember that this is a work in progress and you do not need to define everything now. We can lift, shift and adjust through the course of time.

We already specified one in the url (`:orderId`), so we will use that. But we also need to bring in the `HeaderData`. We could use more parameters, but our method signature could eventually get very long.

Let's build an incoming data interface to specify this information. We will fully build this out in Step 8.4, but for now, we can specify its full class path:

```
SwiftOtter\OrderExport\Api\Data\IncomingHeaderDataInterface
```

What about a return type? We need to also generate a response data interface:

```
SwiftOtter\OrderExport\Api\Data\ResponseInterface
```

This will be populated in Step 8.5.

Two *important* steps are left:

- In PHPStorm, type `/** [enter]` to create the docblock.
- Next, swap out the shortened class paths in the docblock with the full names, like this (yes, it does get quite long, but it wouldn't be Magento if it wasn't long):

```
/**
   * @param int $orderId
   * @param
\SwiftOtter\OrderExport\Api\Data\IncomingHeaderDataInterface
$incomingHeaderData
   * @return \SwiftOtter\OrderExport\Api\Data\ResponseInterface
   */
public function execute(int $orderId, IncomingHeaderDataInterface
$incomingHeaderData): ResponseInterface;
```

If we leave these interfaces short, Magento will throw an error like `"IncomingHeaderDataInterface does not exist"`. That is because Magento's reflection system has trouble understanding **`use`** clauses.

Step 8.3: Build the endpoint class

Video Link

```
app/code/SwiftOtter/OrderExport/Model/Endpoint/Export.php
\SwiftOtter\OrderExport\Model\Endpoint\Export
```

This is the class that will replace the controller that we already built. When a request matching the definition in `etc/webapi.xml` hits the Magento API, this class and its method will be executed.

The logic is the same, the inputs and the outputs are a little different.

Once you have created this class, don't forget to specify the preference for the `ExportInterface` to the `Export` class. Otherwise, you will get an error message about not being able to instantiate an interface (a duh! moment for sure).

Now, copy the code from our Run controller into the export endpoint. We will need to change this a little—for example, we need to build out our header data, or get it from somewhere?

It is a good time to go to etc/di.xml and copy the commented code or remove this comment:

```
<type name="SwiftOtter\OrderExport\ViewModel\OrderDetails">
   <plugin
        name="InjectAdminToken"
        type="SwiftOtter\OrderExport\Plugin\InjectAdminToken" />
</type>
```

If you are building, line-by-line, this module for yourself, please copy `SwiftOtter\OrderExport\Plugin\InjectAdminToken` into your project. This class is something you will want to always take with you as a developer, or at least the token-generated parts. In order for the WebAPI to understand *who it is* that is creating this request, you must generate a token for this particular admin user. **`self`** authentication does NOT work for the adminhtml area.

This plugin modifies the output of our ViewModel (a great example that plugins can also change business logic of our own modules) and enables sending information to the web API.

Once you have completed this, feel free to try submitting an export from the frontend. You should get an error having to do with not being able to instantiate the `IncomingHeaderDataInterface`. This is normal and represents the next step in what we need to do.

Step 8.4: Build the incoming header data interface and class

Video Link

```
app/code/SwiftOtter/OrderExport/Api/Data/IncomingHeaderDataInterface.
php
SwiftOtter\OrderExport\Api\Data\IncomingHeaderDataInterface
```

This interface represents the data that is passed from the order export form and associated Javascript into the Web API. Before, we utilized the RequestInterface->getParam() method. Now, we have an interface and class dedicated to transporting this data—and that is what we are building now.

The advantage is we get to force types and even automatically scaffold objects before it gets to our business logic in the endpoint.

How does it do this?

Magento looks at the incoming data and compares the keys to the specified interfaces (starting with the method defined in `etc/webapi.xml`). In this example, Magento sees that `$orderId` was specified in the URL. But, `$incomingHeaderData` is not explicitly defined in the XML file—instead it is defined in `ExportInterface`.

Magento now looks for a `incomingHeaderData` POST data key. Magento tries to instantiate the `IncomingHeaderDataInterface` (which will be mapped to a class in `etc/di.xml`). Magento then maps the keys in the `incomingHeaderData` array to `set` methods. For example the array key, `ship_date`, Magento will call `setShipDate` on the class.

You can better understand how this mapping works once you create this class. Set a breakpoint in one of the set methods and trigger the API by submitting the order export.

Let's get it done.

Let's start by taking the method definitions that we have already defined in `Model/HeaderData.php` and bring them into this interface.

Next, we change out `\DateTime` references. Again, Magento's API is not compatible with the `\DateTime` class and you will get an error if you do have it specified in a method's arguments or return type. Change these references to a string.

Create the implementation

```
app/code/SwiftOtter/OrderExport/Model/IncomingHeaderData.php
SwiftOtter\OrderExport\Model\IncomingHeaderData
```

Let's now create a new model that will implement this interface. Feel free to copy the `HeaderData` class as a template.

Joseph!! Wait! Why do I need to change out the `\DateTime` references? Can't I just swap this out through our code? Of course you can. But, we are not in the business of quick and dirty. We want to learn the best way to accomplish this. Through this process, you will learn a neat trick.

Once you change `\DateTime` references to string, and add a preference to `etc/di.xml`, you are *almost* done.

You might be seeing a sticking point. We sent `HeaderData` to our `Orchestrator`, but we just created a new class `IncomingHeaderData`—and the two are incompatible. How do we translate the `IncomingHeaderData` class into `HeaderData`?

How about we use a `HeaderDataFactory`? You can add a `getHeaderData` method to the `IncomingHeaderDataInterface` and `IncomingHeaderData` class—which creates a new

`HeaderData` class, populates the values (along with converting the `string` into a `\DateTime`) and returns it.

This might seem confusing because we don't want a `\DateTime` class associated with an interface that Magento uses for the web API. There is an exception to that rule: Magento must not call a method that uses `\DateTime`. In this case, `IncomingHeaderDataInterface` is populated (with the set methods) during a POST request. At no time does Magento ever call any get methods on this interface—in our code, we call it, but not Magento. As such, we are free to use whatever types we wish to use.

Update the Export endpoint class

While you are at it, go back to the Export endpoint class and change out the `$headerData` definition to `getHeaderData` from the `IncomingHeaderDataInterface`. Wow, this is so clean!

Step 8.5: Create the response implementation and class

Video Link

```
app/code/SwiftOtter/OrderExport/Api/Data/ResponseInterface.php
SwiftOtter\OrderExport\Api\Data\ResponseInterface
```

We are onto the home stretch. This interface is very easy—we need to take the array keys we specified in our orchestrator and create get and set methods for them, like `getSuccess` and `setSuccess`.

Make sure you create the appropriate docblocks for these.

Note: you can add array brackets after an interface to denote an array of this type. This is a very powerful feature. Like:

```
SwiftOtter\MyProject\Api\Data\MyModelInterface[]
```

Create the implementation

```
app/code/SwiftOtter/OrderExport/Model/ResponseDetails.php
SwiftOtter\OrderExport\Model\ResponseDetails
```

Create two private variables and then use PHPStorm's code generation feature (Alt+Enter) to create getters and setters.

Update the Export endpoint class

Add a factory for the ResponseDetails class, set the values and return them.

What resources can I safely trust?

- Magento core code: I believe this is the most under-utilized resource for personal development. As you see in this project, I constantly refer back to it. Magento built this application and Magento's example is the best. Even if you are an ace, senior developer, you will still rely on reference material—and the source code. The difference is how long it will take you to figure out a problem and resolve it.

- Magento DevDocs: this is a fantastic resource. It provides direction on many subjects—but, there are a few exceptions.

- Magento StackExchange: answers from users with 2k reputation and more are usually acceptable. Still, hold even these answers with a grain of salt.

- Mage2.tv

- Alan Storm

- Atwix

- Mage2.pro

- m.academy

- IntegerNet