DUMPSBOSS.

Adobe Commerce Developer Professional

Adobe AD0-E717

Version Demo

Total Demo Questions: 9

Total Premium Questions: 97

Buy Premium PDF

https://dumpsboss.co

support@dumpsboss.co

support@dumpsboss.co
dumpsboss.co

Topic Break Down

Topic	No. of Questions
Topic 1, Working with Adobe Commerce architecture	42
Topic 2, Working with Adobe Commerce database	15
Topic 3, Customizing Adobe Commerce	14
Topic 4, Working with Adobe Commerce UI	21
Topic 5, Working with Adobe Commerce checkout	4
Topic 6, Working with Adobe Commerce APIs	1
Total	97

QUESTION NO: 1

Which two attribute input types does Magento already have by default? (Choose two.)

A. Multiple Select

B. Text Field

C. Geographic coordinate

D. Numeric Field

ANSWER: A B

Explanation:

Multiple Select and **Text Field** are standard Adobe Commerce product attribute input types available when creating an attribute in the Admin. Multiple Select is used when a product can be assigned more than one value from a predefined list of attribute options. For example, a clothing product might have several applicable materials, features, or compatible device models. The selected values are stored as attribute option values and can be used in catalog filtering, layered navigation when configured appropriately, product data entry, and rules where the attribute is eligible.

Text Field is the default single-line free-form attribute input type. It is appropriate for concise textual product data that does not require the administrator to choose from a controlled list. Adobe Commerce also provides input-validation settings for text fields, allowing implementations to constrain entered values to formats such as decimal numbers, integers, URLs, emails, dates, or letters. This validation capability does not create a separate numeric input type; it remains a Text Field attribute with the relevant validation applied. Adobe's attribute-creation documentation lists both Multiple Select and Text Field among the available Catalog Input Type for Store Owner values. See the Adobe Commerce documentation for creating product attributes and the product attributes overview.

QUESTION NO: 2

A module contains several related cron jobs. The jobs must be declared individually, but they must share cron-group settings such as schedule generation and separate-process execution.

Which two files should the developer configure? (Choose two.)

A. `etc/crontab.xml`

B. `etc/cron_groups.xml`

C. `etc/di.xml`

D. `app/etc/env.php`

ANSWER: A B

Explanation:

Individual cron jobs are declared in `etc/crontab.xml`, where each job specifies its name, instance, method, schedule, and group. Shared operational settings for the named group are declared in `etc/cron_groups.xml`. The group configuration can define schedule-generation behavior, history cleanup, missed-job handling, and whether jobs run in separate processes.

`di.xml` is used for dependency-injection configuration and does not define cron schedules or group settings. `env.php` contains deployment-specific application configuration, not module cron declarations.

QUESTION NO: 3

A developer is tasked with creating a new feature in an Adobe Commerce Cloud project. The developer decides to create an integration environment for a better development process.

Which Cloud CLI for Commerce command would the developer use?

A. magento-cloud environment:branch

B. magento-cloud create:environment-branch

C. magento-cloud environment:create:branch environment-name >

ANSWER: A

Explanation:

The command `magento-cloud environment:branch <environment-name> <parent-environment-id>` is used to create a new environment by branching from an existing parent environment in an Adobe Commerce on cloud infrastructure project. This is the appropriate workflow for feature development because the resulting branch is an Integration environment, providing an isolated deployment target for implementing and validating the new feature without directly changing the parent environment. The first argument supplies the name for the new environment, while the second identifies the source environment from which code, configuration, and data relationships are branched. For example, a developer can branch from the project's integration environment into a feature-specific environment, push commits to that branch, and allow the cloud build and deployment process to validate the work. Adobe's Cloud CLI documentation lists `environment:branch` as the environment-management command for this operation, and Adobe's environment guidance describes Integration environments as the place to develop and test features before merging them onward. See the Cloud CLI branching documentation and the development and deployment workflow documentation.

QUESTION NO: 4

A merchant wants to include taxes In an Adobe Commerce store. Which option can have a tax class assigned to it?

A. Order

B. Shipping

C. Category

ANSWER: B

Explanation:

Shipping is correct because Adobe Commerce supports assigning a specific tax class to shipping charges. In the store configuration, the *Tax Class for Shipping* setting determines the tax class that Commerce applies when calculating tax on the shipping amount. This allows a merchant to treat freight or delivery charges according to the applicable tax rules for the store's jurisdiction, independently of the tax classes assigned to catalog products or customer groups.

Tax classes are central to Commerce tax calculation: tax rules combine customer tax classes, product tax classes, and tax rates. Shipping is included through its dedicated configuration field, which selects one of the defined product tax classes for the shipping amount. After the shipping tax class is selected and the relevant tax rule and rates are configured, Commerce can calculate and display tax for shipping during checkout and in orders, subject to the merchant's broader tax settings. Adobe documents this setting under the Tax configuration options, including whether tax is applied to shipping amounts and which tax class is used for those amounts. See the Adobe Commerce Tax configuration reference and the Adobe Commerce tax rules documentation.

QUESTION NO: 5

How can a developer prioritize a plugin's execution, if possible?

A. The developer can use sortOrder property by specifying a lower value than the target plugin.

B. The developer can use sortOrder property by specifying a higher value than the target plugin.

C. This cannot be achieved as the plugins are always executed by their module's load order in app/etc/config.php file.

ANSWER: A

Explanation:

The developer can use sortOrder property by specifying a lower value than the target plugin. Adobe Commerce plugins are declared in `di.xml`, and the `sortOrder` attribute controls their relative invocation sequence when more than one plugin intercepts the same public method. For plugin types that run before the intercepted method, a plugin with a lower numeric `sortOrder` is invoked ahead of one with a higher value. This gives a developer a supported, explicit mechanism for prioritizing interception logic instead of relying on module loading order. For example, assigning `sortOrder="10"` to a plugin causes its before-method logic to run ahead of another applicable plugin assigned `sortOrder="20"`. The setting belongs on the plugin declaration under the applicable observed type in dependency-injection configuration. Using a deliberate numeric order is particularly important when plugins modify arguments, shared state, or behavior that later interception logic depends on. Adobe's plugin documentation describes `sortOrder` as the configuration used to determine the order in which plugins are called, and its dependency-injection documentation covers defining plugin declarations in `di.xml`. See Adobe Commerce plugins documentation and Adobe Commerce dependency injection configuration documentation.

QUESTION NO: 6

A module adds a configuration section that must be visible only to Admin users granted a custom permission.

Which two configuration changes are required? (Choose two.)

A. Declare the custom ACL resource in `etc/acl.xml`.

B. Reference the custom ACL resource from the section in `etc/adminhtml/system.xml`.

C. Declare the configuration section's default value in `etc/config.xml`.

D. Declare a preference for the configuration section class in `etc/di.xml`.

ANSWER: A B

Explanation:

The module must declare its permission resource in `etc/acl.xml` and assign that resource to the configuration section in `etc/adminhtml/system.xml`. The ACL declaration makes the resource available for role assignment, and the section's `resource` element causes the Admin configuration system to check that permission before displaying or allowing access to the section.

Declaring a default value in `etc/config.xml` does not control Admin authorization. A preference in `di.xml` changes dependency-injection behavior and is not used to restrict access to system configuration sections.

QUESTION NO: 7

A developer must extend the behavior of an existing RequireJS JavaScript component without replacing the original module. The customization should add behavior while preserving the component's original implementation.

Which two actions should the developer take? (Choose two.)

A. Declare the target module and mixin module in `view/frontend/requirejs-config.js`.

B. Create a JavaScript module that returns an extension of the target component.

C. Declare a PHP preference for the target component in `etc/frontend/di.xml`.

D. Edit the deployed target JavaScript file under `pub/static`.

ANSWER: A B

Explanation:

The developer should declare the mixin in the module's `view/frontend/requirejs-config.js` file and create a mixin JavaScript module that receives the target component and returns the extended component. Adobe Commerce loads the mixin when RequireJS resolves the target module, allowing the customization to extend the original behavior without copying or replacing the core file.

Adding a preference in `di.xml` applies to PHP dependency injection and has no effect on browser-side RequireJS modules. Editing a generated file under `pub/static` is not maintainable because static content can be regenerated or redeployed.

QUESTION NO: 8

A developer is creating a class \Vendor\Module\Model\MyModeL How should that class be defined as transient in di.xml?

A.

B.

C.

ANSWER: C

Explanation:

`<type name="\\Vendor\\Module\\Model\\MyModeL" shared="false" />` is the correct DI configuration for a transient class. Adobe Commerce's dependency-injection object manager treats configured types as shared by default. A shared type can have its instance reused when the object manager resolves that dependency during an application request. Setting `shared="false"` disables that shared-instance behavior for the specified type, so the object manager creates a new instance each time the type is requested. This is the appropriate configuration when the class must not be reused, such as when it contains request-specific or mutable state that should remain isolated between consumers. The declaration belongs in the module's applicable `etc/di.xml` configuration scope and must use the class's fully qualified name exactly, including the capitalization shown in the class declaration. Adobe's DI documentation explicitly identifies the `shared` attribute as the mechanism for controlling whether an object is shared, and notes that `false` causes a new instance to be created when requested. See the Adobe Commerce dependency injection documentation and the di.xml configuration reference.

QUESTION NO: 9

A custom REST service receives a `SearchCriteriaInterface` and must retrieve only products with a specified attribute value through the product repository.

Which repository method should the service use?

A. Call `ProductRepositoryInterface::getList($searchCriteria).`

B. Call `ProductRepositoryInterface::get($searchCriteria).`

C. Call `addAttributeToFilter()` on the product repository.

D. Call `ProductRepositoryInterface::save($searchCriteria).`

ANSWER: A

Explanation:

The service should call `ProductRepositoryInterface::getList()` and pass the constructed `SearchCriteriaInterface`. Repository `getList()` methods are designed to apply search criteria, including filters, filter groups, sorting, and pagination, while returning a search-results object.

`get()` retrieves one product by SKU and does not accept search criteria. Calling `addAttributeToFilter()` is a collection-level approach; it bypasses the repository contract required by this service design. `save()` persists product data and does not retrieve matching products.