PassQueen
a.k.a dumpsqueen.com

Salesforce Accredited B2B Commerce Developer

Salesforce B2B-Commerce-Developer

Version Demo

Total Demo Questions: 34

Total Premium Questions: 344

Buy Premium PDF

https://passqueen.com

support@passqueen.com

QUESTION NO: 1

Which wire adapter should a developer use to retrieve metadata about a specific object?

A. getObjectMetadata

B. getObjectInfo

C. getObject

D. getObjectDescribe

ANSWER: B

Explanation:

`getObjectInfo` is the Lightning Data Service wire adapter used to retrieve metadata for one Salesforce object. A Lightning Web Component imports it from `lightning/uiObjectInfoApi` and supplies an `objectApiName`, commonly imported from `@salesforce/schema`. The returned object-info response provides the metadata needed for UI and business logic that must adapt to an object's configuration. This includes properties such as the object label and API name, key prefix, field definitions, child relationships, record type information, and theme details. For example, a component can use this response to determine available record types or inspect field-level metadata before presenting a dynamic form. Because this adapter is reactive, the component receives updated data when its reactive parameters change. Salesforce documents `getObjectInfo` as the object metadata API for Lightning Web Components; see the getObjectInfo wire adapter reference and the Object Info API reference.

QUESTION NO: 2

A developer is working on an existing checkout implementation built against the Lightning Web Runtime (LWR) and wants to implement a custom child checkout component to modify out-of-the-box functionality.

Which interface must the developer implement for the child component?

A. CheckoutSavable

B. Checkoutoutinterface

C. CustomCheckout

D. CheckoutStep

ANSWER: D

Explanation:

`CheckoutStep` is the required interface for a custom child checkout component in an LWR-based B2B or B2C Commerce storefront. Salesforce's composable checkout architecture uses a parent checkout component and child step components. A child component that participates in this framework implements `CheckoutStep` from the `commerce/checkout` module so it can integrate with the checkout lifecycle and coordinate with the checkout data provider. This enables the component to receive checkout state, participate in stage actions, validate its inputs, save data when appropriate, and communicate errors or completion status through the expected checkout mechanisms. The interface builds on the shared checkout-component capabilities supplied by `UseCheckoutComponent`, allowing a custom implementation to behave consistently with standard checkout steps while replacing or extending out-of-the-box behavior. This contract is important because checkout orchestration depends on child steps exposing the expected methods and state-handling behavior rather than merely rendering a Lightning web component. Salesforce documents custom LWR checkout development and identifies `CheckoutStep` as the interface used for these child checkout components. See Create a Custom Checkout Component for an LWR Store and the CheckoutStep interface reference.

QUESTION NO: 3

Which two statement are true for Mass Order (2 answers)

A. Mass Order pricing is done via a batch job.

B. Mass order works with thedefault wishlists

C. The variation product is leveraged for SKUs.

D. Mass Order is mobile ready with the ccrz templates.

ANSWER: A C

Explanation:

Mass Order pricing is done via a batch job because Mass Order is designed to support high-volume ordering workflows without calculating every price synchronously as a shopper enters a quantity. The batch-based pricing process evaluates the submitted mass-order data and applies the applicable B2B Commerce pricing configuration, making this feature appropriate for users who need to work with large sets of items or quantities. The variation product is leveraged for SKUs because a variation product represents a product family whose purchasable variants are distinguished by attribute combinations, such as color, size, material, or packaging. Each selectable variant maps to a specific SKU, enabling a Mass Order process to identify the exact purchasable item when the user supplies or selects the relevant SKU information. This is consistent with B2B Commerce's product-catalog model, in which product and SKU data must be structured so that the storefront and ordering features can resolve the intended sellable item. For Salesforce B2B Commerce implementation context, see the Salesforce B2B Commerce developer documentation and Salesforce's B2B Commerce overview.

QUESTION NO: 4

A developer needs to implement a business rule around shipping: Anything over a certain amount must be sent through freight at a fixed cost and cannot be

sent through standard shipping channels. Which two considerations should the developer consider?

A. A class will need to implement the sfdc_commerce.CartShippingCharges interface

B. The implementing class cannot call out to another service

C. A class will need to implement the sfdc_checkout.CartShippingCharges interface

D. The implementing class may need to contain logical branching

ANSWER: A D

Explanation:

"A class will need to implement the sfdc_commerce.CartShippingCharges interface" is correct because B2B Commerce provides this extension point for replacing the standard shipping-charge calculation with merchant-specific Apex logic. The implementation can inspect cart data, including the applicable merchandise total, and return the freight charge when the configured threshold is exceeded. This makes the shipping rule part of the Commerce cart-calculation flow rather than a disconnected post-processing step.

"The implementing class may need to contain logical branching" is also correct. The requirement has at least two distinct outcomes: carts at or below the threshold can continue through the normal shipping calculation, while carts above the threshold must receive the fixed freight charge and be prevented from using standard-shipping treatment. Conditional logic in the custom shipping-charges implementation is therefore needed to evaluate the threshold and apply the appropriate outcome. The interface-based approach allows that logic to be executed consistently as Commerce calculates cart charges. Salesforce's B2B Commerce extensibility guidance and Apex reference describe using Commerce APIs and Apex extension points to tailor checkout and cart behavior: Salesforce Commerce Developer Guide and sfdc_commerce Apex Namespace Reference.

QUESTION NO: 5

What is the difference between Gross Layout Overrides and Subscriber Templates?

A. Gross Layout Overrides allow modification to CSS of a page, while Subscriber Templates allows for modification of the entire page including header and footer.

B. Subscriber Templates allows for modification of the header, the footer and the content in between them. Gross Layout Overrides only allow for modification of the header and footer.

C. Subscriber Templates allow for modification of the header and the footer, while Gross Layout Overrides allow for modification everything inside the header and footer.

D. Gross Layout Overrides allow for the modification of the footer, while Subscriber Templates allow for modification of everything inside the header and footer.

ANSWER: C

Explanation:

Subscriber Templates allow for modification of the header and the footer, while Gross Layout Overrides allow for modification everything inside the header and footer. In the legacy Salesforce B2B Commerce (CloudCraze) storefront architecture, a Subscriber Template establishes the shared outer page chrome for a subscriber storefront. This shared structure includes areas such as the header and footer, which are intended to remain consistent as customers navigate between storefront pages. A Gross Layout Override is used when a page needs a different layout for its central or body region without replacing that common storefront chrome. It therefore controls the content rendered between the header and footer, enabling page-specific presentation while retaining the subscriber-level framing around it. This separation supports reuse and maintainability: branding and persistent navigation belong in the Subscriber Template, whereas the main page experience belongs in the Gross Layout Override. Salesforce's B2B Commerce developer learning resources describe the storefront's configurable templates, layouts, and components: Salesforce Trailhead: B2B Commerce for Developers. For current platform context and developer resources, see Salesforce B2B Commerce Developer Documentation.

QUESTION NO: 6

Which three file extensions are allowed in a Lightning Web Component folder?

A. .js-meta.xml

B. .html

C. .js

D. .gif

E. .jar

ANSWER: A B C

Explanation:

A Lightning Web Component is packaged as a component bundle: a folder whose files share the same base name and have supported extensions. The `.html` file supplies the component template, defining the markup rendered by the component. The `.js` file contains the JavaScript module, including the component class, reactive properties, event handlers, and other client-side behavior. The `.js-meta.xml` file is the configuration file that defines metadata for the component, such as its API version and the Salesforce targets where administrators can expose it, including Lightning pages or Experience Builder pages. These three file types form the standard minimal structure used to build and deploy a Lightning Web Component. File extensions are case-sensitive in the expected bundle convention, so the JavaScript extension is written as `.js`. Salesforce documents the supported files in a component bundle and explains that the component folder and its files must use the same base name. See Salesforce Developer Guide: Component Folder and Lightning Component Configuration File Reference.

QUESTION NO: 7

What are three ways to test the value of Page Label on any Salesforce B2B

Commerce Community Page? (3 answers)

A. Access the source HTML for the page viathe browser developer tools.

B. Execute CCRZ.pagevars.pageLabels['PAGE_LABEL_NAME'] in the JavaScript console.

C. Execute CCRZ.processPageLabelMap('PAGE_LABEL_NAME') in the JavaScript console.

D. Enable the 'display page label names' in cc admin.

E. Execute (('PAGE_LABEL_NAME')) in the JavaScript console

ANSWER: B C D

Explanation:

Salesforce B2B Commerce storefronts expose page-label data through the global `CCRZ` JavaScript namespace, so entering `CCRZ.pagevars.pageLabels['PAGE_LABEL_NAME']` in the browser console is a direct way to inspect the resolved value made available to the current page. `CCRZ.processPageLabelMap('PAGE_LABEL_NAME')` is also intended for processing and retrieving a named page label from the page-label map, making it useful for checking the configured label value during storefront testing. In addition, enabling *display page label names* in CC Admin provides a visual testing mode in the storefront: label names are displayed with their rendered values, allowing an administrator or developer to identify which configured label supplies visible text without manually querying every label in the console. These approaches cover both developer-console inspection and storefront-level visual verification, which is helpful when confirming configuration changes, translations, or label substitutions. Page labels are a B2B Commerce storefront configuration feature and are managed as part of the B2B Commerce administration and development model. See Salesforce's B2B Commerce Developer Guide and Salesforce B2B Commerce documentation for the platform documentation.

QUESTION NO: 8

Which two Salesforce B2B Commerce visualforce pages must be enabled at a Salesforce Community level to make the out of the box SEO functionality available? (2 answers)

A. CCSizeIndex

B. SizeMap

C. CCCatSiteMap

D. ProductMap

ANSWER: A C

Explanation:

CCSizeIndex and **CCCatSiteMap** are the Visualforce pages that must be made available to the Salesforce Community for the legacy, out-of-the-box B2B Commerce SEO sitemap capability. CCSizeIndex acts as the sitemap-index endpoint: it provides the crawler-facing index that points search engines toward the available sitemap content. CCCatSiteMap supplies the category sitemap content, allowing category URLs to be discovered and crawled. Enabling both pages at the community level is necessary because Salesforce Experience Cloud sites restrict Visualforce-page access unless the pages are explicitly enabled for the community. Once available, these endpoints support the sitemap-based discovery process used by search engines, which is especially important for commerce catalogs whose navigable category pages may otherwise be difficult for crawlers to locate consistently. Salesforce describes how Visualforce pages can be enabled for an Experience Cloud site in its Experience Cloud Visualforce page documentation. The sitemap-index and sitemap relationship follows the standard described in the Sitemaps Protocol.

QUESTION NO: 9

Which two steps are necessary to enable Salesforce B2B Commerce

logging in the managed package?

A. Ensure you save a value in the Logging Token input field in the Global Settings section of CC Admin.

B. Turn On theCheckbox "Cloudcraze Logging" in CC Admin.

C. Ensure the value saved in the Logging token field is appended to the ccLog query parameter.

D. Set a cookie with the Id of the user accessing the storefront in CC Admin

ANSWER: A C

Explanation:

Salesforce B2B Commerce managed-package logging is enabled through a token-based mechanism intended to allow targeted troubleshooting without leaving verbose logging enabled for all storefront traffic. First, an administrator must enter and save a Logging Token in the Global Settings area of CC Admin. The token can be a chosen string and acts as the authorization value for requesting diagnostic logging. Next, that exact saved value must be included in the `ccLog` query-string parameter on the storefront URL being investigated. For example, if the configured token is `debug`, visiting a URL such as `https://my-storefront.example/?ccLog=debug` enables logging for that request/page context. Matching the URL parameter to the saved token is essential: the stored token controls access, while the query parameter activates logging for the specific storefront session or request being debugged. This approach supports controlled, temporary diagnostics in a production-like storefront environment. For Salesforce B2B Commerce implementation and administration resources, see Salesforce Help: B2B Commerce and Salesforce B2B Commerce Developer Guide.

QUESTION NO: 10

A query containing a subquery is executed. What is appended to the subquery name

as part of its transformation by default in Salesforce B2B Commerce?

A. A subscriber-supplied token

B. "__ccrz"

C. The "*" symbol

D. The letter "S"

ANSWER: D

Explanation:

The letter "S" is appended to the subquery name by default during Salesforce B2B Commerce query transformation. This behavior supports the framework's handling of subquery relationship names when it prepares the query for execution. B2B Commerce's query-processing conventions account for Salesforce object relationships, where a parent-to-child query uses a child relationship name in its nested SELECT clause. Adding the suffix helps the framework produce the expected transformed relationship reference for the subquery rather than treating it as an ordinary top-level object query.

This is relevant when working with the legacy B2B Commerce (CloudCraze) APIs and query framework because developers must account for framework-generated query transformations instead of assuming that submitted query text is always executed verbatim. Understanding the default suffix is particularly useful when diagnosing a query that includes nested relationship data or when matching query results to transformed query definitions. Salesforce describes parent-to-child relationship queries and nested SELECT syntax in its SOQL documentation: Relationship Queries. Additional B2B Commerce developer resources are available from Salesforce B2B Commerce Developer Documentation.

QUESTION NO: 11

Which two components in a B2B store template should a developer use to customize a storefront page?

A. My Lists

B. Product List

C. Order List

D. Address List

ANSWER: A B

Explanation:

My Lists and **Product List** are storefront components intended for adding product-list functionality to a B2B Commerce store template. My Lists provides buyers with a page or area where they can work with their saved product lists. This supports common B2B purchasing behavior, such as maintaining recurring-order lists, favorite-item lists, or lists organized for a particular job, location, or purchaser. A developer can place and configure this component as part of a tailored storefront experience rather than building the underlying list-management user interface from scratch.

Product List is used to render the products that belong to a selected product list. It provides the product-focused presentation needed to let a buyer view items in a list and continue the purchasing flow from those items. Used together, My Lists supplies the buyer's list context and Product List supplies the associated product display, making them complementary choices for a customized storefront page.

This use of reusable storefront components aligns with Salesforce's B2B Commerce approach of composing buyer experiences from configurable page components and extending them when business requirements require it. See Salesforce's B2B Commerce Developer Guide and the B2B Commerce Basics learning module for B2B storefront concepts.

QUESTION NO: 12

A user wants to leverage a three columnlayout on a page. The user also wants to move the mini-cart widget from the right to the center column. How can this requirement be fulfilled?

A. Gross Layout Override

B. Subscriber Template

C. Page Include

D. HandleBar Template Override

ANSWER: A

Explanation:

Gross Layout Override is correct because this requirement changes the overall structural layout of the page, rather than simply changing the markup or behavior inside an existing content region. A gross layout defines the high-level page framework, including the available columns and the placement of major regions or widgets. Creating a Gross Layout Override enables a developer to replace the standard page structure with a three-column arrangement and assign the mini-cart widget to the center column. This approach is appropriate when the desired result requires both a different column configuration and relocation of a widget from one layout region to another. The override provides control over how the page's broad visual areas are assembled while retaining the B2B Commerce page's intended widget-based composition model. Salesforce B2B Commerce development uses templates, layouts, and extensions to tailor storefront experiences; structural changes should be implemented at the level that owns the page layout. See Salesforce's Commerce developer documentation and the B2B Commerce Basics Trailhead module for platform architecture and storefront customization context.

QUESTION NO: 13

Northern Trail Outfitters (NTO) has acquired a company and is looking to manage product data across the org seamlessly. The company has a governance policy to not install any tool or use third-party API applications to export or import the data into Salesforce. However, users have access to Salesforce CLI.

Which set of tasks must a developer perform whento export data from Salesforce or import data into Salesforce?

A. sfdx force:data:bulk:export -Product2 -all 0 and

B. sfdx force:data;tree:export -Product2 -all q and

C. sfdx force:tree:data:export -q "SELECT Id, Name FROM Product2" -u ® and

D. sfdx force:data:tree:export -q "SELECT Id, Name FROM Product2" -u "" and sfdx force:data:tree:import -f Product2.json -u ""

ANSWER: D

Explanation:

`sfdx force:data:tree:export -q "SELECT Id, Name FROM Product2" -u "<your username>"` and `sfdx force:data:tree:import -f Product2.json -u "<your username>"` is correct because Salesforce CLI provides native data tree commands that operate through the authenticated Salesforce org connection, without requiring installation of an external import/export application. The export command runs the specified SOQL query against `Product2` and serializes the returned records into Salesforce data tree JSON. This format includes the metadata needed by the tree import process, such as record type information and reference identifiers where applicable.

The import command then reads the generated JSON file and creates the represented Product2 records in the target org identified by the supplied username or org alias. Data tree commands are particularly useful for migration scenarios involving a manageable volume of records and can also preserve relationships when a query includes related records. A developer should authenticate the CLI to each relevant org, use a SOQL query that selects the needed fields, confirm the generated file name and location, and supply that file to the import command. Salesforce documents the data tree export and import commands in the Salesforce CLI Command Reference; the underlying API behavior is described in the sObject Tree REST API documentation.

QUESTION NO: 14

A Developer created a custom field that a project wants to expose on a given page.

How does the Developer ensure that the field is available to display on a given page?

A. Override the Service Class that the page uses and update the ServiceManagementin CCAdmin for the given storefront to use this new Service Class.

B. Override the Logic Class that the page uses and update the Service Management inCCAdmin for the given storefront to use this new Service Class

C. Create a new Service Classthat the page uses and update the Service Managementin CCAdmin for the given storefront to use this new Service Class

D. Create anew Logic Class that the page uses and update the Service Managementin CCAdmin for the given storefront to use this new Service Class

ANSWER: A

Explanation:

Override the Service Class that the page uses and update the ServiceManagementin CCAdmin for the given storefront to use this new Service Class. This is the appropriate approach because, in Salesforce B2B Commerce, a page's service class is responsible for assembling the data that the page needs. To expose a newly created custom field, the developer extends or overrides the service class associated with that page and updates its data-retrieval or response-building behavior to

include the field. The storefront must then be configured in CCAdmin Service Management to use the customized service implementation; otherwise, the page continues to invoke its existing service class and does not receive the additional field data. This configuration is storefront-specific, which allows a customized implementation to be enabled where needed without changing the underlying page architecture. After the service returns the custom-field value, the page's presentation layer can bind to and render that value. Salesforce's B2B Commerce developer documentation describes service classes as the extension point for supplying page data and identifies CCAdmin service management as the mechanism for associating service implementations with storefront behavior. See the Salesforce B2B Commerce Service Classes documentation and the B2B Commerce extension documentation.

QUESTION NO: 15

Which three decorators can be used in Lightning Web Components?

A. @api

B. @track

C. @wire

D. @class

E. @import

ANSWER: A B C

Explanation:

`@api`, `@track`, and `@wire` are decorators provided by the `lwc` module for Lightning Web Components. Use `@api` to expose a component field or method as part of its public API, allowing a parent component to set the property or invoke the method. Use `@wire` to provision reactive data from a Lightning Data Service wire adapter or an Apex method; when reactive parameters change, the wire service can provide updated data. `@track` can be used to observe mutations to fields that hold plain JavaScript objects or arrays, so changes to accessed nested properties can trigger rerendering. Although primitive component fields are reactive without `@track`, the decorator remains supported for the object-and-array observation use case. These decorators are imported explicitly, for example: `import { api, track, wire } from 'lwc';`. Salesforce documents public properties and methods through `@api`, wire service data provisioning through `@wire`, and the current reactivity behavior, including the limited situations where `@track` is needed. See Lightning Web Components Decorators and Reactivity for Fields, Objects, and Arrays.

QUESTION NO: 16

Numerous flags ... have adirectimpact on the result set provided by the Global

API's. What Global API Data-Sizing convention flag prevents an API request from

propagating to further requests when provided as a Boolean parameter with a value of

true?

A. ccrz.ccAPI.SZ_REL

B. ccrz.ccAPI.SZ_ASSC

C. ccrz.ccAPISizing.ASSC

D. ccrz.ccAPISizing.REL

ANSWER: B

Explanation:

`ccrz.ccAPI.SZ_ASSC` is the data-sizing convention flag used to control whether a sizing selection is associated with downstream API calls. When this Boolean flag is supplied with a value of `true`, the sizing block applies only to the immediate Global API request rather than being propagated to additional requests that the original operation might invoke. This lets a developer request a specific amount of data for one operation while allowing dependent or subsequent operations to use their normal, independently defined sizing behavior. For example, a cart retrieval can use a large sizing block for the cart itself without forcing that same sizing configuration onto related calls made during processing. This supports predictable response payloads and helps avoid unnecessarily large data graphs in chained Global API operations. The `ASSC` naming reflects the association control for sizing conventions, making it the appropriate Boolean parameter for stopping sizing propagation. Salesforce's B2B Commerce documentation describes Global API sizing conventions and the API framework used by managed-package storefront implementations. See the Salesforce B2B Commerce Developer Guide and the Salesforce B2B Commerce documentation.

QUESTION NO: 17

A developer is trying to troubleshoot why a field is not displaying on the Product Detail Page. What should be typed in the Developer Tools Console in the browser to view the fields available for the Product Detail Page?

A. CCRZ.productSearchView

B. CCRZ.cartView

C. CCRZ.productDetailModel

D. CCRZ.productDetailView

ANSWER: C

Explanation:

`CCRZ.productDetailModel` is the browser-side CloudCraze B2B Commerce model that represents the product currently being rendered on a Product Detail Page. Entering this expression in the browser Developer Tools Console exposes the model object and its available properties, allowing the developer to inspect the product data returned to the page. This is the appropriate troubleshooting starting point when a field is expected but does not appear, because the developer can verify whether the field is present in the page's product model before investigating presentation-layer configuration. The model can be expanded in the console to review its attributes and nested data, helping distinguish a missing data value from a display or template issue. Salesforce's B2B Commerce documentation describes the platform's storefront capabilities and product-related configuration context at Salesforce Help: B2B Commerce Overview. For broader developer documentation and supported Commerce implementation resources, see Salesforce Developer Documentation: Commerce.

QUESTION NO: 18

Which two steps are necessary to enable Salesforce B2B Commerce

logging in the managed package?

A. Ensure you save a value in the Logging Token input field in the Global Settings section of CC Admin.

B. Turn On the Checkbox "Cloudcraze Logging" in CC Admin.

C. Ensure the value saved in the Logging token field is appended to the ccLog query parameter.

D. Set a cookie with the Id of the user accessing the storefront in CC Admin

ANSWER: A C

Explanation:

Managed-package logging in the legacy Salesforce B2B Commerce (CloudCraze) implementation is deliberately protected by a logging token so that detailed diagnostic output is generated only for explicitly authorized requests. Administrators first configure this shared token in the *Logging Token* field in the Global Settings area of CC Admin. Saving the value makes the

token available to the managed package's logging mechanism and establishes the value that must be presented when logging is requested.

Logging is then enabled for an individual storefront request by appending that same configured value to the `ccLog` query parameter. For example, if the configured token is `myToken`, a request can include `?ccLog=myToken` (or add `&ccLog=myToken` when the URL already has parameters). The package compares the request parameter with the configured token before emitting its diagnostic logging. This two-part configuration avoids permanently exposing verbose application logging to all storefront traffic while allowing a developer or administrator to troubleshoot a specific browser session or request flow.

For broader Salesforce Commerce documentation and implementation context, see Salesforce Commerce Developer Documentation and Salesforce Help: B2B Commerce.

QUESTION NO: 19

How does a project implement the process to persist payment information datain the

Checkout flow for Salesforce B2B Commerce version 4.2 and beyond?

A. Trigger a remote action when the process payment button is selected to capture the payment.

B. Trigger a remote action to store the payment information in the URL query parameters.

C. Trigger the processPayment event and pass in the payment information object as an argument.

D. Trigger the externalprocessedPayment and pass in the payment information object as an argument.

ANSWER: C

Explanation:

Trigger the processPayment event and pass in the payment information object as an argument. In Salesforce B2B Commerce version 4.2 and later, checkout is designed to use its event-based payment extension point so that payment details are supplied to, and persisted within, the checkout process in the expected lifecycle. The payment-information object carries the payment data required by the configured payment implementation, allowing the checkout flow to coordinate payment processing without placing sensitive details in client-visible URL state. This approach also keeps the payment integration aligned with the checkout component contract: the checkout flow receives a payment request through its designated event, then invokes the appropriate configured payment behavior as part of completing the order transaction. Payment integrations should generally use tokenized or provider-managed payment data rather than handling raw card data directly, helping the implementation meet security and PCI responsibilities. Salesforce's current Commerce checkout documentation describes checkout as the process that coordinates payment and order placement, while its security guidance emphasizes protecting sensitive payment information. See Salesforce Commerce Checkout APIs and Salesforce Privacy and Security information.

QUESTION NO: 20

What is default behavior for how the Salesforce B2B Commerce Global APIs transform Salesforce data?

A. Fields names are returned using the Salesforce naming convention.

B. Fields names are returned with „c." prepended in their name.

C. Fields names are returned with a lowercase first letter, camelcase convention

D. Fields names can be mapped to any naming convention desired

ANSWER: C

Explanation:

Fields names are returned with a lowercase first letter, camelcase convention is correct because Salesforce B2B Commerce Global APIs expose Salesforce record data through an API-oriented JSON representation rather than returning Salesforce field API names verbatim. In the default transformation, the initial character of a field name is lowercased and the resulting property is expressed in camel case. For example, a Salesforce-style field such as `AccountNumber` is represented as `accountNumber` in the transformed API payload. This convention makes response properties consistent with common JSON and JavaScript client-development practices while allowing the Global APIs to provide a simplified commerce-facing data contract. The transformation is part of the standard behavior of the Global API layer, so consumers should use these lower-initial camel-case property names when reading response data and constructing integrations around the default payload shape. Salesforce's B2B Commerce developer documentation describes the Global API framework and its data model conventions in the B2B Commerce Global APIs guide. Additional platform context for developing B2B Commerce integrations is available in the B2B Commerce developer guide.

QUESTION NO: 21

Numerous flags ... have a direct impact on the result set provided by the Global

API's. What Global API Data-Sizing convention flag prevents an API request from

propagating to further requests when provided as a Boolean parameter with a value of

true?

A. ccrz.ccAPI.SZ_REL

B. ccrz.ccAPI.SZ_ASSC

C. ccrz.ccAPISizing.ASSC

D. ccrz.ccAPISizing.REL

ANSWER: B

Explanation:

`ccrz.ccAPI.SZ_ASSC` is the Global API data-sizing convention flag used to control association traversal. When it is supplied as a Boolean parameter with a value of `true`, the API treats associated-data expansion as bounded and prevents the request from continuing to propagate through additional associated requests. This is important in the legacy B2B Commerce (CloudCraze) Global API model because product, category, and other commerce records can be connected through associations that would otherwise cause further data retrieval. Applying this sizing flag lets an integration request the required level of associated information while avoiding unbounded traversal, excessive response payloads, and unnecessary server work. The flag therefore belongs in the request's sizing/parameter map when the caller needs to stop follow-on association requests rather than merely shape a field list or relationship result. Salesforce's developer documentation describes how API integrations use request parameters and response structures to control retrieved data; see the Salesforce REST API Developer Guide and the Apex REST request reference for the general request-parameter and API-processing concepts underlying this behavior.

QUESTION NO: 22

Which three files are required for adeployable Lightning Web Component called displayMyData that will fetch and display data?

A. displayMyData.css

B. displayMyData.js-meta.xml

C. displayMyData.js

D. displayMyDataController.cls

E. displayMyData.html

ANSWER: B C E

Explanation:

A deployable Lightning Web Component bundle named `displayMyData` requires `displayMyData.js`, `displayMyData.html`, and `displayMyData.js-meta.xml`. The JavaScript file supplies the component class and the logic that retrieves data, such as calling an Apex method or using a Lightning Data Service wire adapter. The HTML file is the component template, defining the markup that renders the retrieved data for users. The metadata configuration file is required in every Lightning Web Component bundle; it defines metadata such as the API version and, when applicable, the targets where the component can be exposed in Lightning pages or other supported contexts. These three files form the essential deployable structure for a component that both obtains and presents data. Salesforce documents the required component configuration file and its role in defining a component bundle in the Lightning Web Components configuration file reference. The relationship between a JavaScript component class and its HTML template is described in the Create Lightning Web Components guide.

QUESTION NO: 23

Salesforce B2B Commerce natively provides a robots.txt file, however, a customer

can also create its own version.Which three scenarios are valid reasons for customer to

create their own robots.txt file? (3 answers)

A. The customer wants to reference multiple storefront sitemap indexes in a single robots.txt file

B. The customer wants to reference a custom sitemap index.

C. The customer wants to have multiple robot.txt files in a single Salesforce Community.

D. The customer's store is not located at the rootof their domain.

E. robot.txt only works if there is one storefront in the org

ANSWER: A B D

Explanation:

Creating a custom robots.txt file is appropriate when a merchant needs control beyond the standard B2B Commerce-generated sitemap reference. Referencing multiple storefront sitemap indexes in one robots.txt file gives search engines a single, domain-level discovery point for sitemap indexes that represent separate storefronts. This is especially useful when a shared domain serves more than one storefront and the merchant wants crawler discovery to be coordinated from the required robots.txt location.

Referencing a custom sitemap index is also a valid reason because the merchant may need an index that includes sitemap sources or a sitemap organization tailored to its own SEO architecture. The `Sitemap:` directive in robots.txt supports declaring sitemap locations, allowing search engines to discover the intended index efficiently.

A custom file is additionally appropriate when the store is not hosted at the root of its domain. Robots.txt is requested at the domain root, so a merchant whose commerce experience is deployed beneath a path must manage the root-level crawler directives and sitemap references in a way that correctly points search engines to that store's content. Google documents the root-location behavior and sitemap directives for robots.txt in its robots.txt introduction and sitemap guidance.

QUESTION NO: 24

A developer has made a component with a lightning combobox in the follow! markup. To handle changes on the combobox, what should replace < CHANGE FVENT > ?



A. {event:handleChange}

B. javascript:void(0);handleChange();

C. {handleChange()}

D. {handleChange}

ANSWER: D

Explanation:

`{handleChange}` is the correct replacement because Lightning Web Components bind declarative event listeners by assigning a JavaScript method reference to the event attribute. In this markup, the `lightning-combobox` component's `onchange` attribute should therefore be written as `onchange={handleChange}`. The component's JavaScript class defines `handleChange(event)`, and the framework calls that method whenever the user selects a different combobox value.

The event object supplied to the handler provides the selected value through `event.detail.value`. For example, a handler can assign that value to a reactive component property, use it to update displayed data, or invoke further application logic. Curly braces in an LWC template identify a property or method on the component instance; the method is referenced without parentheses so that it is registered as the callback rather than executed while the template is rendered. Salesforce documents `lightning-combobox` as firing a change event when its value changes, and LWC event handling uses the `on[eventname]` convention to attach handlers. See the lightning-combobox component reference and Salesforce's LWC event-listener guidance.

QUESTION NO: 25

Which three actions are applicable when extendinga default Salesforce B2B Commerce page via a page include? (3 answers)

A. Create a Service Class override to query the new page include.

B. Create the VisualForce page you wish to include to the Salesforce b2B Commerce page.

C. Prepend "c." to the name ofthe page referenced in the configuration setting.

D. Create a configuration setting for enabling the page include and assigning the new page include via CC admin.

E. Build and activate a new configuration cache setting via CC admin.

ANSWER: B D E

Explanation:

Extending a standard Salesforce B2B Commerce storefront page through a page include begins by creating the Visualforce page that supplies the additional markup, components, and any required controller logic. That Visualforce page is the reusable extension content that Commerce inserts at the configured location in the standard page lifecycle. Next, an administrator creates a Commerce configuration setting that both enables the include and maps the new Visualforce page to the intended page and include position. This configuration is managed in CC Admin, allowing the storefront to resolve the include dynamically without modifying the base Commerce page. Finally, the configuration cache must be built and activated in CC Admin. Commerce uses the active configuration cache at runtime, so activating the rebuilt cache makes the new page-include assignment available to the storefront. Together, these steps provide the supported configuration-driven customization approach: define the Visualforce content, register it through Commerce configuration, and publish the resulting configuration cache. Salesforce's Commerce documentation describes configuration-based storefront customization and administration in the B2B Commerce documentation; the platform's Visualforce fundamentals are covered in the Visualforce Developer Guide.

QUESTION NO: 26

A developer is implementing an Inventory class for checkout. All the error states have been handled and now the developer needs to take the next step to indicate that inventory is available for all of the items and amounts in the cart. What should the next step be?

A. Return TRUE

B. Return sfde_checkout.InventoryStatus. SUCCESS

C. Return sfdc_checkout.IntegrationStatus. Status. SUCCESS

D. Return sfdc_checkout.InventoryStatus.Status. SUCCESS

ANSWER: D

Explanation:

Return `sfdc_checkout.InventoryStatus.Status.SUCCESS`. A custom checkout inventory integration communicates its outcome to the Salesforce checkout orchestration through the status value returned by the inventory-validation implementation. Once the implementation has evaluated every cart item and requested quantity, and no inventory-related failure condition applies, the success status is the explicit signal that inventory validation completed successfully. This allows checkout processing to continue to subsequent stages rather than treating the result as an unresolved validation state or an inventory exception.

The value belongs to `InventoryStatus` in the `sfdc_checkout` namespace because the result being reported is specifically inventory availability. Its nested `Status.SUCCESS` value is the platform-defined enum constant for a successful inventory check, rather than a Boolean return value. Salesforce's checkout integration model uses these typed status objects so integrations can provide a consistent result to the managed checkout flow, including status and any applicable messages when needed. Review the Salesforce Commerce checkout integration documentation at Checkout Integrations and the Apex namespace reference at sfdc_checkout Namespace.

QUESTION NO: 27

A developer is debugging a flow and needs to watch all the variables changing as the checkout process is executed, but nothing is displaying. Which two features did the developer forget to enable?

A. Set up a debug tog to show the details of what is executed

B. Show the details of what is executed and render flow in Lightning Runtime

C. Run the latest version of each flow called by subtle w elements

D. Show the details of what is executed and render flow in Lightning Experience.

ANSWER: B D

Explanation:

“Show the details of what is executed and render flow in Lightning Runtime” and “Show the details of what is executed and render flow in Lightning Experience.” identify the debug settings needed to make execution information visible while the checkout flow runs. Flow Builder’s Debug capability can display a detailed execution trace, including the flow path taken and the values supplied to and produced by flow elements. Enabling the setting to show execution details is what exposes the runtime information needed to observe variable changes rather than merely launching the flow. Rendering the flow in the applicable Lightning-based runtime provides the supported UI context for exercising a screen-driven checkout process during debugging. In practice, the developer selects the debug option that shows what is executed and uses the relevant Lightning rendering context for the flow being tested. This lets the developer follow the checkout interaction while reviewing the flow’s execution details and variable state. Salesforce documents the Debug button and its available run options in Debug Flows. For the broader Flow Builder development model and runtime behavior, see Salesforce Flow documentation.

QUESTION NO: 28

A Developer is writing unit tests for a subscriber service that is called by a Salesforce B2B Commerce page. Which three testing practices are appropriate? (3 answers)

A. Create the records required by the test.

B. Use an @testSetup method for common test data.

C. Use Test.startTest and Test.stopTest around the operation under test when appropriate.

D. Depend on the active storefront configuration and production catalog records.

E. Use SeeAllData=true for all Commerce service tests.

ANSWER: A B C

Explanation:

Test methods should create the data required for the scenario rather than depend on records already present in an org. Shared test records can be created once with an `@testSetup` method, which makes the setup reusable while keeping individual test methods focused on their own assertions. `Test.startTest()` and `Test.stopTest()` should be used around the operation under test when the Developer needs a fresh governor-limit context or needs asynchronous work to complete before assertions are made. Tests should verify observable outcomes, such as returned values, created records, or handled errors. See Salesforce guidance on test setup methods and Test.startTest and Test.stopTest.

QUESTION NO: 29

Which three actions are applicable when modifying the number of steps required in the Salesforce Commerce Checkout flow? (3 answers)

A. Perform a template override on the Checkout page.

B. Add a page include to the checkout page.

C. Build and activate a new configuration cache setting via CC admin.

D. Set the value of the configuration setting defined as CO.useDef to TRUE

E. Set the value of the configuration setting defined as CO.overrideFlow to TRUE.

ANSWER: B C E

Explanation:

The supported approach for changing the legacy Salesforce B2B Commerce checkout flow combines a checkout-page extension point with configuration-cache controls. **Add a page include to the checkout page.** provides the extension location where custom checkout-flow behavior can be introduced without directly replacing the base page implementation. This lets the implementation participate in the checkout experience while retaining the platform's standard page structure.

Build and activate a new configuration cache setting via CC admin. is required because checkout-flow behavior is driven by Commerce configuration values that are loaded through the Commerce configuration cache. Creating the setting alone is insufficient; activation makes the configured value available to the running storefront.

Set the value of the configuration setting defined as CO.overrideFlow to TRUE. explicitly enables the custom checkout-flow override. With that override enabled, the checkout can use the customized flow definition and therefore accommodate a changed number of steps. These actions reflect the configuration-oriented extensibility model used by the legacy Commerce/CloudCraze checkout implementation. For broader Salesforce B2B Commerce architecture and development context, see Salesforce Trailhead: B2B Commerce Basics and the Salesforce Commerce Developer Documentation.

QUESTION NO: 30

Northern Trail Outfitters (NTO) exports Order Summary data from its org. A developer launches Data Loader, selects Order Summary, clicks the Select All Fields button, and clicks Finish. Custom fields are defined in the Cart and Order Summary objects and successfully mapped from Cart to Order Summary during checkout. However, all three custom fields in the Order Summary are empty in the export file.

What is the most likely cause?

A. There was a misspelling in one of the custom fields.

B. The Cart to Order action does not support the mapping of custom fields.

C. The developer does not have access to the fields.

D. The developer can export Order Summary records only by using Data Export Service

ANSWER: C

Explanation:

The developer does not have access to the fields. Data Loader honors the permissions of the user who runs the export, including field-level security. Clicking *Select All Fields* selects the fields that are available to that user; it does not override field visibility. If the developer lacks Read access to the custom fields on Order Summary through their profile or permission set, those field values will not be available for export. This can appear especially confusing when the checkout mapping itself is known to work: the Cart-to-Order Summary mapping and the exporting user's field permissions are separate concerns. An administrator should verify that the custom Order Summary fields are visible and have Read access for the developer's user profile or an assigned permission set, then repeat the export. Salesforce describes field-level security as controlling whether users can view and edit individual fields, and Data Loader operations run in the context of the authenticated user. See Salesforce's field-level security documentation and the Data Loader guide.

QUESTION NO: 31

A developer needs to make a call to a long running web service which is critical to finalizing their checkout process. Which three items should the developer consider in their implementation?

A. A new CORS entry may need to be created in Setup

B. An Apex method returning a Continuation will need to be created

C. Requests to the service should be brokered to prevent limit exceptions

D. A new Remote Site may need to be created in Setup

ANSWER: A B C

Explanation:

“A new CORS entry may need to be created in Setup,” “An Apex method returning a Continuation will need to be created,” and “Requests to the service should be brokered to prevent limit exceptions” are the appropriate considerations for a long-running, checkout-critical integration. If browser-side storefront code needs to access a service hosted on another origin, Salesforce CORS configuration can be required so that the browser permits the cross-origin interaction. Salesforce describes the configuration process and the origin-based nature of CORS in its CORS documentation.

A Continuation lets Apex initiate a long-running HTTP request without tying up the request-processing thread while the remote service responds. The associated callback processes the response after it returns, making this pattern suitable where checkout must wait for an external decision while still handling an extended callout duration appropriately. Salesforce documents the asynchronous request-and-callback model in its Apex Continuations guide.

Because checkout traffic can produce simultaneous requests, brokering requests provides a controlled integration boundary. It can coordinate, queue, consolidate, or otherwise regulate requests so that bursts do not unnecessarily consume platform resources or cause governor-limit-related failures. This is especially important when the external service is essential to completing the transaction.

QUESTION NO: 32

Which of these is a key pattern leveraged when building Lightning Web Components? 39m 36s

A. Composition

B. Inventory

C. Juggling

D. Normalization

ANSWER: A

Explanation:

Composition is a key pattern for building Lightning Web Components. In LWC, an application is structured as a hierarchy of small, focused components, where a parent component includes and coordinates child components through their HTML tags. This approach encourages reusable UI units: a child can encapsulate a specific responsibility and expose public properties or methods when interaction with its parent is needed. Components can also communicate upward through custom events, enabling a clear and maintainable flow of data and behavior across the component tree. Composition makes it easier to develop complex B2B Commerce storefront experiences from modular pieces, such as product displays, filters, cart controls, and navigation elements, without placing all logic and markup in one large component. Salesforce’s LWC documentation describes component composition as nesting components within other components and explains how public properties and events support parent-child communication. See Compose Components and Create and Dispatch Events.

QUESTION NO: 33

A developer exports data from an org on a standard entity which has a custom attribute. When they launch Data Loader, select the entity, click the Select All Fields button and click Finish, the custom field they added called MyCustomField_c has no values and no column header in the CSV file. What is the root cause?

A. The user needs to install a specific Zulu JDK that is recommended by Salesforce.

B. A mapping file was not used when the data was loaded in

C. The user does not have access to the field

D. The user has rights to the field but there are no values in it

ANSWER: C

Explanation:

The correct root cause is: “The user does not have access to the field.” Data Loader operates in the context of the authenticated Salesforce user and respects that user’s field-level security. When Data Loader retrieves the available fields for an export, fields that the user is not permitted to read are not made available for selection. Consequently, the inaccessible custom field is omitted entirely from the generated export: it has neither a CSV column header nor any exported values. Access to a custom field is controlled through field-level security settings assigned by profiles and permission sets. An administrator should grant the user Read access to the custom field on the relevant object, then have the user sign in to Data Loader again and repeat the export. Once read access is available, selecting all fields includes the custom field’s API name as a CSV header, with values returned for records where the field is populated. Salesforce documents that field-level security controls user access to individual fields, independently of page-layout visibility. See Salesforce Help: Control Access to Fields and Salesforce Data Loader Developer Guide: Export Data.

QUESTION NO: 34

The ccUtil apex class in Salesforce B2B Commerce provides numerous utility

functions that can be leveraged in subscriber classes.

What are two ways to check the input or return data of the Global API's? (2 answers)

A. ccrz.ccUtil.isNotEmpty(Map < String, Object >) and ccrz.ccUtil.isNotEmpty(List < Object >)

B. ccrz.ccUtil.isNotValid(Map < String, Object >) and ccrz.ccUtil.isNotValid(List < Object >)

C. ccrz.ccUtil.isValid(Map < String, Object >) and ccrz.ccUtil.isValid(List < Object >)

D. ccrz.ccUtil.isEmpty(Map < String, Object >) and ccrz.ccUtil.isEmpty(List < Object >)

ANSWER: A D

Explanation:

`ccrz.ccUtil.isNotEmpty(Map<String, Object>)` together with `ccrz.ccUtil.isNotEmpty(List<Object>)`, and `ccrz.ccUtil.isEmpty(Map<String, Object>)` together with `ccrz.ccUtil.isEmpty(List<Object>)`, are the B2B Commerce utility methods intended for safely evaluating Global API request and response collections. Subscriber implementations commonly receive API data as a parameter map and return result data in maps or lists. Checking those collections before reading values, iterating records, or applying business logic prevents null-reference failures and allows the subscriber to take an appropriate path when an API call supplies no data. `isNotEmpty` provides a concise positive guard when processing should continue only when a collection exists and contains entries. `isEmpty` provides the complementary guard for fallback behavior, initialization, or an empty-result response when a map or list is null or contains no entries. These utility checks align with Apex collection handling, where maps and lists expose emptiness semantics and should be evaluated defensively before use. Salesforce’s Apex reference documents collection behavior and methods for List and Map types.