Salesforce.PDII.v2026-03-10.q74

Exam Code:	PDII
Exam Name:	Platform Developer II
Certification Provider:	Salesforce
Free Question Number:	74
Version:	v2026-03-10
# of views:	122
# of Questions views:	740
https://www.freepdfdumps.com/Salesforce.PDII.v2026-03-10.q74.html	

NEW QUESTION: 1
Refer to the test method below:
Java
@isTest
static void testAccountUpdate() {
Account acct = new Account(Name = 'Test');
acct.Integration_Updated__c = false;
insert acct;
CalloutUtil.sendAccountUpdate(acct.Id);
Account acctAfter = [SELECT Id, Integration_Updated__c FROM Account WHERE Id = :acct.Id][0]; System.assert(true, acctAfter.Integration_Updated__c);
}
The test method calls a web service that updates an external system with Account information and sets the Account's Integration_Updated__c checkbox to True when it completes. The test fails to execute and exits with an error: "Methods defined as TestMethod do not support Web service callouts." What is the optimal way to fix this?
A. Add if (!Test.isRunningTest()) around CalloutUtil.sendAccountUpdate.
B. Add Test.startTest() before and Test.stopTest() after CalloutUtil.sendAccountUpdate.
C. Add Test.startTest() before and Test.setMock and Test.stopTest() after CalloutUtil.sendAccountUpdate.
D. Add Test.startTest() and Test.setMock before and Test.stopTest() after CalloutUtil.sendAccountUpdate.
Answer: (SHOW ANSWER)
Salesforce enforces a strict restriction: Actual network callouts are prohibited during unit tests. This is to ensure that tests are deterministic, fast, and do not rely on the availability or state of external third-party systems. When the testing engine encounters a System.Http.send() or a web service call without a mock, it throws the error: "Methods

defined as TestMethod do no1t support Web servi2ce callouts."34 To resolve this, the developer must provide a Mock Implementation. By using Test.setMock() (Option D), the developer instructs the Apex runtime to intercept any callouts and return a pre-defined response instead of attempting a real connection. The mock class5 must implement either the HttpCalloutMock interface (for REST) or the WebServiceMock interface (for SOAP). Furthermore, the call to the mock and the callout method should be wrapped in Test.startTest() and Test.
stopTest().6
* Test.startTest(): Resets governor limits, providing a fresh context for the specific logic being tested.7
* Test.stopTest(): Forces any asynchronous processing (often used in cal8louts, such as @future or Queueable) to complete before the next line of code executes.
In the provided code, Test.setMock must be called before CalloutUtil.sendAccountUpdate for the platform to know which mock to use. Once Test.stopTest() is reached, the mock response is processed, the checkbox is updated, and the subsequent SOQL query and assertion will correctly see the updated data. Option A is a poor practice because it skips the logic entirely, resulting in 0% code coverage for the integration logic.

NEW QUESTION: 2
Which technique can run custom logic when a Lightning web component is loaded?
A. Use an <aura:handler> init event to call a function.
B. Use the connectedCallback() method.
C. Use the renderedCallback() method.
D. Call $A.enqueueAction and pass in the method to call.
Answer: (SHOW ANSWER)
In Lightning Web Components (LWC), the component lifecycle is managed by standard Web Component lifecycle hooks. To run logic exactly once when the component is inserted into the Document Object Model (DOM), a developer should use the connectedCallback() method (Option B).
The connectedCallback() is the LWC equivalent of Aura's init event. It is the ideal place to:
* Perform initial data fetching (via imperative Apex).
* Initialize internal state or variables.
* Subscribe to a Lightning Message Service (LMS) channel.
* Establish communication with the parent component.
Option A and D are Aura-specific syntax and are not valid in LWC. Option C (renderedCallback()) runs every time the component finishes a render cycle. Because a component can re-render many times (whenever a reactive property changes), placing initialization logic in renderedCallback can lead to performance issues or infinite loops if not guarded carefully. Therefore, connectedCallback() is the standard, most efficient way to handle "on load" logic in LWC.

NEW QUESTION: 3
A developer wrote a class named AccountHistoryManager that relies on field history tracking. The class has a static method called getAccountHistory that takes in an Account as a parameter and returns a list of associated AccountHistory object records. The following test fails:
Java
@isTest
public static void testAccountHistory(){
Account a = new Account(name = 'test');
insert a;
a.name = a.name + '1';
update a;
List<AccountHistory> ahList = AccountHistoryManager.getAccountHistory(a);
System.assert(ahList.size() > 0);
}
What should be done to make this test pass?
A. Use @isTest(SeeAllData=true) to see historical data from the org and query for AccountHistory records.
B. Use Test.isRunningTest() in getAccountHistory() to conditionally return fake AccountHistory records.
C. The test method should be deleted since this code cannot be tested.
D. Create AccountHistory records manually in the test setup and write a query to get them.
Answer: B (LEAVE A REPLY)
A significant challenge in Salesforce unit testing is that system-generated records, such as those in the AccountHistory or OpportunityHistory objects, are not created during the execution of a test method, even if field history tracking is enabled in the organization. Because these records are read-only and managed by the system, a developer cannot manually insert them via DML within a test setup. Consequently, the query in the getAccountHistory method will always return an empty list in a test context, causing the System.assert to fail.
The optimal programmatic solution to this limitation is to implement a "mocking" strategy using Test.
isRunningTest(). By modifying the production code within getAccountHistory, the developer can check if the code is currently being executed by a unit test. If it is, the method can return a manually constructed list of AccountHistory records (stored in memory but not inserted into the database) to satisfy the test's requirements. This allows the test to verify the logic that processes the history data without needing the system to actually generate historical records. While SeeAllData=true (Option A) would allow the test to see real data in the org, it is considered a poor practice because it makes the test dependent on the specific state of the organization's data, leading to brittle tests that may fail in different environments.12

==========34

NEW QUESTION: 4
Given the following containment hierarchy:
HTML
<template>
<c-my-child-components></c-my-child-components>
</template>
What is the correct way to communicate the new value of a property named "passthrough" to my-parent- component if the property is defined within my-child-component?
A. let cEvent = new CustomEvent('passthrough', { detail: 'this.passthrough' });
this.dispatchEvent(cEvent);
B. let cEvent = new CustomEvent('passthrough'); this.dispatchEvent(cEvent);
C. let cEvent = new CustomEvent('passthrough', { detail: this.passthrough });
this.dispatchEvent(cEvent);
D. let cEvent = new CustomEvent($passthrough); this.dispatchEvent(cEvent);
Answer: (SHOW ANSWER)
In Lightning Web Components (LWC), data flows "down" via properties and "up" via events. When a child component needs to communicate a change in a property or state to its parent, it must dispatch a CustomEvent. To pass specific data-such as the new value of the passthrough property-along with the event, the developer must use the detail property within the event initialization object.
Option C is the correct syntax. It creates a new CustomEvent named 'passthrough' and assigns the current value of the component's property (this.passthrough) to the detail key. The parent component can then listen for this event (using onpassthrough={handleEvent}) and access the value via event.detail. Option A is incorrect because it wraps the variable in quotes, passing the literal string "this.passthrough" instead of the actual data. Option B creates an event but fails to include the data payload, meaning the parent would know an event occurred but wouldn't receive the new value. Option D uses incorrect syntax for event naming and variable referencing. Using the standard CustomEvent constructor with the detail property is the platform- standard way to ensure robust, typed data communication between component layers in the Shadow DOM.

NEW QUESTION: 5
A company has a native iOS order placement app that needs to connect to Salesforce to retrieve consolidated information from many different objects in a JSON format. Which is the optimal method to implement this in Salesforce?
A. Apex SOAP web service
B. Apex SOAP callout
C. Apex REST callout
D. Apex REST web service

Answer: D (LEAVE A REPLY)
When an external application needs to request data from Salesforce, you must expose an endpoint. Since the requirement specifically mentions JSON format and a native mobile app (iOS), an Apex REST web service (Option D) is the optimal choice. REST (Representational State Transfer) is the industry-standard architecture for mobile-to-cloud integrations because it is lightweight, uses standard HTTP methods, and natively supports JSON.
By using the @RestResource and @HttpGet annotations in an Apex class, a developer can create a custom endpoint that performs complex logic, such as querying multiple objects (Accounts, Orders, Line Items), and returns a single, consolidated JSON response. This is far more efficient than the standard REST API if the mobile app would otherwise need to make multiple sequential calls to gather the same information.
Options B and C are incorrect because "callouts" are used when Salesforce requests data from an external system, not the other way around. Option A (SOAP) is a heavier, XML-based protocol that is less efficient for mobile development and does not use the JSON format natively. Therefore, a custom Apex REST service provides the best performance and flexibility for mobile integrations.

NEW QUESTION: 6
Universal Containers is leading a development team that follows the source-driven development approach in Salesforce. As part of their continuous integration and delivery (CI/CD) process, they need to automatically deploy changes to multiple environments, including sandbox and production. Which mechanism or tool would best support their CI/CD pipeline in source-driven development?
A. Salesforce CLI with Salesforce DX
B. Change Sets
C. Salesforce Extensions for Visual Studio Code
D. Ant Migration Tool
Answer: A (LEAVE A REPLY)
Source-driven development shifts the "source of truth" from the Salesforce org to a version control system (like Git). Salesforce DX (Developer Experience) was specifically designed to support this paradigm by introducing the Salesforce CLI (Command Line Interface). The Salesforce CLI is the primary engine for modern CI/CD pipelines because it allows for the scripted creation of scratch orgs, automated testing, and the deployment of source code to any environment (sandboxes, scratch orgs, or production) using simple command-line instructions3.
Unlike traditional tools, the Salesforce CLI supports the "Source" format, which is more granular and easier to manage in version control than the traditional Metadata format used by the Ant Migration Tool. Change Sets (Option B) are manual, UI-driven tools that cannot be automated in a CI/CD pipeline and are strictly org-to- org. Salesforce Extensions for VS Code (Option C) provide a powerful IDE interface for developers, but the extensions

themselves rely on the underlying Salesforce CLI to perform deployments; they are not the automation tool used by the pipeline itself. The Ant Migration Tool (Option D) is an older technology that uses the Metadata API and is generally considered legacy compared to the more flexible and powerful Salesforce DX commands. Therefore, Salesforce CLI with Salesforce DX is the optimal choice for a robust, automated source-driven deployment strategy.

NEW QUESTION: 7
Refer to the following code snippet:
Java
public class LeadController {
public static List<Lead> getFetchLeadList(String searchTerm, Decimal aRevenue) { String safeTerm = '%'+searchTerm.escapeSingleQuotes()+ '%'; return [SELECT Name, Company, AnnualRevenue FROM Lead WHERE AnnualRevenue >= :aRevenue AND Company LIKE :safeTerm LIMIT 20
];
}
}
A developer created a JavaScript function as part of a Lightning web component (LWC) that surfaces information about Leads by wire calling getFetchLeadList when certain criteria are met. Which three changes should the developer implement in the Apex class above to ensure the LWC can display data efficiently while preserving security?1
A. Use the WITH SECURITY_ENFORCED clause within the SOQL query.
B. Implement the with sharing keyword in the class declaration.567
C. Annotate 8the Apex method with @AuraEnabled(Cacheable=true).
D. Implement the without sharing keyword in the class declaration.
E. Annotate the Apex method with @AuraEnabled.
Answer: A,B,C (LEAVE A REPLY)
Compreh17ensive and Detailed 11850 to 250 words of Explanation:
To make an Apex method compatible with a Lightning Web Component's @wire service and ensure it follows security best practices, three specific modifications are required:
* @AuraEnabled(Cacheable=true) (Option C): The @wire service in LWC requires the Apex method to be marked as cacheable. This enables client-side caching via the Lightning Data Service, which significantly improves UI performance by reducing redundant server calls. Note that Cacheable=true is mandatory for @wire but optional for imperative calls.
* with sharing (Option B): In Apex, classes do not enforce sharing rules by default. To ensure the user only sees Leads they have access to according to the organization-wide defaults and sharing model, the class must explicitly use the with sharing keyword.
* WITH SECURITY_ENFORCED (Option A): While with sharing handles record-level access, it does not automatically enforce field-level security (FLS) or object-level security

(CRUD). Adding the WITH SECURITY_ENFORCED clause to the SOQL query ensures that if a user does not have permission to view the AnnualRevenue field, the query will throw an exception rather than exposing protected data.
Options D and E are incorrect because without sharing bypasses security, and a simple @AuraEnabled without cacheable=true is insufficient for the LWC @wire service.

NEW QUESTION: 8
A company recently deployed a Visualforce page with a custom controller that has a data grid of information about Opportunities in the org. Users report that they receive a "Maximum view state size limit" error message under certain conditions. According to Visualforce best practice, which three actions should the developer take to reduce the view state?
A. Refine any SOQL queries to return only data relevant to the page.2021
B. Use the final keyword in the controller for variables that will not change.1617
C. Use the private keyword in the controller for variables.1415
D. Use filters and pagination to reduce the amount of data.1819
E. Use the transient keyword in the Apex controller for variables that do not maintain state.
Answer: (SHOW ANSWER)

NEW QUESTION: 9
The Salesforce admin at Cloud Kicks created a custom object called Region__c to store all postal zip codes in the United States and the Cloud Kicks sales region the zip code belongs to.
Object Name: Region__c
Fields: Zip_Code__c (Text), Region_Name__c (Text)
Cloud Kicks wants a trigger on the Lead to populate the Region based on the Lead's zip code. Which code segment is the most efficient way to fulfill this request?1234
A. 5678
Java
Set<String> zips = new Set<String>();
for(Lead l : Trigger.new) {
if(l.PostalCode != Null) {
zips.add(l.PostalCode);
}
}
for(Lead l : Trigger.new) {
List<Region__c> regions = [SELECT Zip_Code__c, Region_Name__c FROM Region__c WHERE Zip_Code__c IN :zips]; for(Region__c r : regions) { if(l.PostalCode == r.Zip_Code__c) {
B. Region__c = r.Region_Name__c;
}

}
}
C. Java
Set<String> zips = new Set<String>();
for(Lead l : Trigger.new) {
if(l.PostalCode != Null) {
zips.add(l.PostalCode);
}
}
List<Region__c> regions = [SELECT Zip_Code__c, Region_Name__c FROM Region__c WHERE Zip_Code__c IN :zips]; for(Lead l : Trigger.new) { for(Region__c r : regions) { if(l.PostalCode == r.Zip_Code__c) {
D. Region__c = r.Region_Name__c;
}
}
}
E. Java
for(Lead l : Trigger.new) {
Region__c reg = [SELECT Region_Name__c FROM Region__c WHERE Zip_Code__c = :l.
PostalCode];
F. Region__c = reg.Region_Name__c;
}
G. Java
Set<String> zips = new Set<String>();
for(Lead l : Trigger.new) {
if(l.PostalCode != Null) {
zips.add(l.PostalCode);
}
}
List<Region__c> regions = [SELECT Zip_Code__c, Region_Name__c FROM Region__c WHERE Zip_Code__c IN :zips]; Map<String, String> zipMap = new Map<String, String>();
for(Region__c r : regions) { zipMap.put(r.Zip_Code__c, r.Region_Name__c);
}
for(Lead l : Trigger.new) {
if(l.PostalCode != Null) {
H. Region__c = zipMap.get(l.PostalCode);
}
}
Answer: D (LEAVE A REPLY)

To adhere to Salesforce Apex best practices, especially when dealing with triggers, code must be "bulkified." This means the code should efficiently handle hundreds of records at once without hitting platform governor limits, such as the limit of 100 SOQL queries per transaction.
Option D is the most efficient segment because it follows the standard Set-Query-Map pattern.
* Set: It first iterates through all Leads in the trigger to collect unique zip codes into a Set<String>.
* Query: It performs a single SOQL query outside of any loops to fetch only the relevant Region__c records. This avoids the "SOQL in a loop" anti-pattern found in Options A and C, which would fail if the trigger processed more than 100 records.
* Map: It stores the query results in a Map<String, String>. Maps are highly efficient for data retrieval because they offer $O(1)$ search complexity.
* Final Loop: In the final loop, the code uses zipMap.get() to find the region.
In contrast, Option B is bulkified regarding queries but uses a nested loop to match data. If you have 200 leads and 200 regions, the nested loop would perform 40,000 iterations (200×200), which could lead to "Maximum CPU time exceeded" errors for large datasets. Option D eliminates this unnecessary CPU overhead by using the Map to look up the zip code directly, making it the most scalable and performant solution.

NEW QUESTION: 10
Universal Charities (UC) uses Salesforce to collect electronic donations in the form of credit card deductions from individuals and corporations. When a customer service agent enters the credit card information, it must be sent to a 3rd-party payment processor for the donation to be processed. UC uses one payment processor for individuals and a different one for corporations. What should a developer use to store the payment processor settings for the different payment processors, so that their system administrator can modify the settings once they are deployed, if needed?
A. Hierarchy custom setting
B. Custom label
C. Custom metadata
D. List custom setting
Answer: (SHOW ANSWER)
For storing application configurations and integration settings that need to be easily modified by administrators and deployed across environments, Custom Metadata Types are the preferred solution. Unlike Custom Settings (Options A and D), records within a Custom Metadata Type are considered metadata rather than data. This is a critical distinction for the development lifecycle because these records can be included in Change Sets or deployment packages. This eliminates the manual overhead and risk associated with re- creating configuration records in production after a sandbox deployment.

In this scenario, UC needs to manage settings for two different payment processors. A Custom Metadata Type can be created with fields for API endpoints, merchant IDs, and security keys. An administrator can then create and edit the specific records for the "Individual" and "Corporate" processors directly in the Setup menu.
Furthermore, Custom Metadata queries are efficient and do not count against standard SOQL governor limits in many contexts. While Custom Labels (Option B) are useful for translating text, they are not intended for complex, structured configuration data. Hierarchy Custom Settings are designed for user-specific overrides, which is not applicable here. Therefore, Custom Metadata provides the most robust, deployable, and administrator-friendly way to manage external service configurations.

NEW QUESTION: 11
A developer is asked to look into an issue where a scheduled Apex is running into DML limits. Upon investigation, the developer finds that the number of records processed by the scheduled Apex has recently increased to more than 10,000. What should the developer do to eliminate the limit exception error?
A. Implement the Batchable interface.
B. Use platform events.
C. Use the @future annotation.
D. Implement the Queueable interface.
Answer: A (LEAVE A REPLY)
The issue described involves hitting DML limits due to a high volume of records (over 10,000) being processed in a single transaction. In Salesforce, a single synchronous transaction or a standard Scheduled Apex job has strict governor limits on the number of DML statements (150) and the total number of records processed (10,000 for DML rows). To resolve this, the developer should implement the Database.Batchable interface. Batch Apex is specifically designed to handle large data volumes by breaking the processing into smaller, manageable chunks (batches) of records (default 200). Each batch runs within its own transaction context, resetting the governor limits for that specific batch. This prevents the "Too many DML rows" exception. While Queueable (Option D) and @future (Option C) provide asynchronous processing, they are not inherently designed to iterate over large datasets in the same robust, governor-limit-safe manner as Batch Apex, particularly when the record count exceeds the thousands.

NEW QUESTION: 12
An Apex trigger creates a Contract record every time an Opportunity record is marked as Closed and Won. A developer is tasked with preventing Contract records from being created when mass loading historical Opportunities, but the daily users still need the logic to fire. What is the most extendable way to update the Apex trigger to accomplish this?
A. Add the Profile ID of the user who loads the data to the trigger.
B. Add a validation rule to the Contract to prevent creation by the data load user.

C. Use a hierarchy custom setting to skip executing the logic inside the trigger for the user who loads the data.
D. Use a list custom setting to disable the trigger for the user who loads the data.
Answer: C (LEAVE A REPLY)
Managing "Trigger Kill Switches" is a critical requirement for data migration. The most "extendable" and platform-standard approach is using a Hierarchy Custom Setting (Option C).
A hierarchy setting allows for different values at the Organization, Profile, and User levels. A developer can create a checkbox field called Disable_Opportunity_Trigger__c. In the trigger, the code would check: if (MySettings__c.getInstance().Disable_Opportunity_Trigger__c) return;. During a mass data load, the administrator can enable this checkbox specifically for the integration user or the specific administrator performing the load. Because it is a hierarchy, it doesn't affect standard users.
This approach is superior to hard-coding Profile IDs (Option A) because IDs change between environments. It is also better than List Custom Settings (Option D) because it provides the automatic "fallback" logic (User > Profile > Org) that makes management much simpler. Validation rules (Option B) would cause the trigger to fail with an exception, which could stop the entire data load transaction, whereas the Custom Setting approach allows the trigger to "gracefully" skip its logic and allow the Opportunity to be saved without creating a Contract.

NEW QUESTION: 13
Which code snippet processes records in the most memory efficient manner, avoiding governor limits such as
"Apex heap size too large"?
A. Java
Map<Id, Opportunity> opportunities = new Map<Id, Opportunity>([SELECT Id, Amount from Opportunity]); for(Id oppId: opportunities.keySet()){
// perform operation here
}
B. Java
for(Opportunity opp: [SELECT Id, Amount from Opportunity]){
// perform operation here
}
C. Java
List<Opportunity> opportunities = Database.query('SELECT Id, Amount from Opportunity');
for(Opportunity opp: opportunities){
// perform operation here
}
D. Java

List<Opportunity> opportunities = [SELECT Id, Amount from Opportunity]; for(Opportunity opp: opportunities){
// perform operation here
}

Answer: B (LEAVE A REPLY)

When processing large data volumes in Apex, the "Heap Size" limit (6MB for synchronous, 12MB for asynchronous) is a common bottleneck. This limit tracks the amount of memory consumed by objects and variables currently in scope.

Option B utilizes the SOQL For Loop pattern. This is the most memory-efficient approach because it processes records in "chunks" of 200 at a time using internal query locators. Instead of loading the entire result set (potentially thousands of records) into a single collection in memory, the SOQL For Loop iterates through the records without ever holding the full set in the heap. This prevents the "Apex heap size too large" error when dealing with high-volume queries.

In contrast, Options A, C, and D all involve assigning the entire query result to a collection (a Map or List) before the loop begins. This immediately consumes heap space proportional to the number of records and fields retrieved. If the query returns a large number of records, these collections will quickly exceed the heap limit before the first line of the loop even executes. Therefore, for bulk data processing where memory efficiency is the priority, the inline SOQL For Loop (Option B) is the standard best practice.

NEW QUESTION: 14

Universal Containers uses Salesforce to track orders in an Order__c object. The Order__c object has private organization-wide defaults. The Order__c object has a custom field, Quality_Controller__c, that is a Lookup to User and is used to indicate that the specified User is performing quality control on the Order__c. What should be used to automatically give read only access to the User set in the Quality_Controller__c field?

A. User managed sharing

B. Record ownership

C. Apex managed sharing

D. Criteria-based sharing

Answer: C (LEAVE A REPLY)

In Salesforce, when organization-wide defaults (OWD) are set to Private, users only have access to records they own or those shared with them through the role hierarchy or sharing rules. In this scenario, the requirement is to grant access dynamically based on a value in a lookup field (Quality_Controller__c).

Criteria-based sharing rules (Option D) are generally used for field values that are static or belong to 1a predefined set, but they cannot dynamically target a specific user identified in a lookup field on the record itself.2 Apex managed sharing i3s the optimal solution for this requirement. It allows developers to programmatically create sharing records (shares) to grant access to specific users or groups. When a record is created or the

Quality_Controller__c field is updated, an Apex trigger can insert a record into the Order__Share object. This share record would specify the UserOrGroupId as the ID from the lookup field and the AccessLevel as 'Read'.
This approach is highly flexible and ensures that as the quality controller changes, the sharing access is updated accordingly. User managed sharing (Option A), often referred to as manual sharing, requires manual intervention by the record owner and cannot be fully "automatic" in the context of complex business logic without Apex. Therefore, Apex managed sharing provides the necessary automation and precision for record- level security based on dynamic lookup values.

NEW QUESTION: 15
A business requires that every parent record must have a child record. A developer writes an Apex method with two DML statements to insert a parent record and a child record. A validation rule blocks child records from being created. The method uses a try/catch block to handle the DML exception. What should the developer do to ensure the parent always has a child record?
A. Use Database.insert() and set the allOrNone parameter to true.
B. Delete the parent record in the catch statement when an error occurs on the child record DML operation.
C. Set a database savepoint to rollback if there are errors.
D. Use addError() on the parent record if an error occurs on the child record.
Answer: (SHOW ANSWER)
1516
In Salesforce Apex, each DML statement is its own mini-transaction unless specific measures are taken to group them. If a developer performs an insert parent; followed by an insert child;, and the child insertion fails due to a validation rule, the parent record remains in the database. This leads to "orphaned" parent records, violating the bu17siness requireme18nt that every parent must have a child.
To ensure "all-or-nothing" behavior across multiple DML statements, the developer should use Database.
setSavepoint() and Database.rollback(). By setting a savepoint before the parent is inserted, the developer creates a "marker" for the database state. If the child insertion fails and throws an exception, the code enters the catch block. Within the catch block, the developer can call Database.rollback(sp), which reverts the database to the state before the parent was ever inserted. This ensures that either both records are created successfully or neither is created.
Option A is incorrect because allOrNone only applies to the records within a single Database.insert() call; it cannot link the success of two separate DML calls for different objects. Option B is a manual cleanup approach that is less reliable than a system-level rollback. Option D is typically used in triggers to prevent saving, but it doesn't provide the transactional control needed in a custom method with multiple DML steps.

Using savepoints is the standard best practice for managing multi-object transaction atomicity.

NEW QUESTION: 16
A developer created a Lightning web component that allows users to input a text value that is used to search for Accounts by calling an Apex method. The Apex method returns a list of AccountWrappers and is called imperatively from a JavaScript event handler.
Java
01: public class AccountSearcher {
02:
03: public static List<AccountWrapper> search(String term) {
04: List<AccountWrapper> wrappers = getMatchingAccountWrappers(term);
05: return wrappers;
06: }
07:
08:
09: public class AccountWrapper {
10: public Account account { get; set; }
11: public Decimal matchProbability { get; set; }
12: }
13: // ...other methods, including getMatchingAccountWrappers implementation...
14: }
Which two changes should the developer make so the Apex method functions correctly?
A. Add @AuraEnabled to line 01.
B. Add @AuraEnabled to line 09.
C. Add @AuraEnabled to lines 10 and 11.
D. Add @AuraEnabled to line 03.
Answer: C,D (LEAVE A REPLY)
For an Apex method to be reachable by a Lightning Web Component, it must be annotated with
@AuraEnabled. In this code, line 03 is the entry point for the search functionality. Without the
@AuraEnabled annotation on line 03 (Option D), the JavaScript controller will not be able to "see" or invoke the search method, resulting in a runtime error.
Furthermore, because this method returns a custom "Wrapper" class, the platform's serialization engine needs to know which specific properties within that wrapper class should be sent to the client-side JavaScript. By default, the LWC framework does not serialize class members unless they are explicitly marked. Therefore, the developer must add @AuraEnabled to lines 10 and 11 (Option C). Without this, even if the method executes successfully, the account and matchProbability properties will be stripped during the JSON conversion, and the component will receive a list of empty objects.

Adding @AuraEnabled to the class definitions themselves (Lines 01 or 09) is not required; the annotation is only necessary on the specific methods and member variables that participate in the data exchange between Apex and JavaScript. This explicit marking ensures security and efficiency by only exposing the data intended for the user interface.

NEW QUESTION: 17
Universal Containers implements a private sharing model for Convention_Attendee__c. A lookup field Event_Reviewer__c was created. Management wants the event reviewer to automatically gain Read/Write access to every record they are assigned to. What is the best approach?
A. Create an after insert trigger on the Convention Attendee custom object, and use Apex Sharing Reasons and Apex Managed Sharing.
B. Create a before insert trigger on the Convention Attendee custom object, and use Apex Sharing Reasons and Apex Managed Sharing.
C. Create criteria-based sharing rules on the Convention Attendee custom object to share the records with the Event Reviewers.
D. Create a criteria-based sharing rule on the Convention Attendee custom object to share the records with a group of Event Reviewers.
Answer: A (LEAVE A REPLY)
In a private sharing model, access is restricted to the owner and those above them in the hierarchy. When a specific user is identified in a lookup field (like Event_Reviewer__c), standard sharing rules cannot dynamically grant access to that specific individual because sharing rules only target Roles, Public Groups, or Territories.1 The solution is Apex Managed Sharing (Option A). By using an after insert (and likely after update) trigger, the developer can pro2grammatically create records in the object's "Share" table (e.g., Convention_Attendee__Share). Each share record specifies the UserOrGroupId (from the lookup field), the AccessLevel ('Edit' for Read/Write), and a RowCause (Apex Sharing Reason).
The "After" trigger is required because the ID of the Convention_Attendee__c record must exist before a Share record can be associated with it. "Before" triggers (Option B) are used for field updates on the record itself, not for creating related sharing records. Options C and D are not viable because they cannot dynamically reference a User ID stored within a

field on the record itself. Apex Managed Sharing provides the most granular and automated way to handle this dynamic security requirement.

NEW QUESTION: 18
Universal Containers (UC) has enabled the translation workbench and has translated picklist values. UC has a custom multi-select picklist field, Products__c, on the Account object that allows sales reps to specify which of UC's products an Account already has. A developer is tasked with writing an Apex method that retrieves Account records, including the Products__c field. What should the developer do to ensure the value of Products__c is in the current user's language?40
A. Use toLabel(Products__c) in the fields list of the SOQL que41ry.
B. Call the translate() method on each record in the SOQL result list.
C. Set the locale on each record in the SOQL result list.
D. Use the locale clause in the SOQL query.
Answer: A (LEAVE A REPLY)
When a Salesforce organization uses the Translation Workbench, picklist values can be translated into multiple languages. By default, a SOQL query returns the "Master" or API value of a picklist field, which is usually in English. If a developer needs the returned data to reflect the translated labels based on the current user's language settings, they must use the toLabel() function within the SOQL SELECT statement.
Using SELECT toLabel(Products__c) FROM Account (Option A) tells the Salesforce query engine to look up the translated value for each picklist entry in the result set before returning it to Apex. This is particularly useful for UI components or reports where the user expects to see values in their native tongue. It works for both standard picklists and multi-select picklists.
Other options are incorrect because they do not exist as standard Apex features. There is no translate() method on SObject records (Option B), and Apex does not allow you to manually "set the locale" on individual records to trigger translation (Option C). There is also no "locale clause" in SOQL syntax (Option D). Using toLabel() is the platform-standard, most efficient way to handle localized data retrieval directly within the data access layer.

NEW QUESTION: 19
A developer created a Lightning web component that uses a lightning-record-edit-form to collect information about Leads. Users complain that they only see one error message at a time about their input when trying to save a Lead record. Which best practice should the developer use to perform the validations on more than one field, thus allowing more than one error message to be displayed simultaneously?
A. Apex REST
B. Client-side validation
C. Next Best Action

D. Custom validation rules

Answer: B (LEAVE A REPLY)

When using lightning-record-edit-form, server-side validation rules (Option D) typically return errors one at a time as the database engine encounters them, or as a single combined toast message that can be difficult to parse. To provide a superi1or user experience where multiple fields are validated simultaneously before the data even reaches the server, the develo2per shou3ld implement Client-side validation (Option B). In the component's JavaScript controller, the developer can intercept the onsubmit event. By iterating through all the lightning-input-field or standard lightning-input elements, the developer can programmatically check for various conditions (e.g., custom regex patterns, conditional logic between fields, or range checks). Each field can then be marked with a custom error message using the reportValidity() or setCustomValidity() methods. This allows the UI to highlight all invalid fields at once, providing immediate, comprehensive feedback to the user. This approach reduces unnecessary server round-trips and ensures that the user can correct all issues in a single pass before successfully submitting the record.

NEW QUESTION: 20

Which method should be used to convert a Date to a String in the current user's locale?

A. Date.parse

B. String.format

C. String.valueOf

D. Date.format

Answer: D (LEAVE A REPLY)

Handling date and time formatting across different countries is a common requirement in global Salesforce orgs. For example, a user in the US expects MM/DD/YYYY, while a user in the UK expects DD/MM/YYYY.

The Date.format() method (Option D) is specifically designed to handle this localization automatically.

When called on a Date instance, it converts the value into a String based on the locale settings of the logged- in user. This is the most efficient and localized approach as it requires no manual mapping of date patterns.

Option A (Date.parse) is the inverse; it converts a localized String into a Date object.

Option C (String.

valueOf) returns a standard, non-localized format (YYYY-MM-DD), which is used for system logging or API payloads but is often confusing for end-users. Option B (String.format) is used for variable substitution within a string (e.g., "Hello {0}") and does not inherently handle date localization. By using the native .format() method, developers ensure a consistent and familiar user experience for employees worldwide.

NEW QUESTION: 21

There is an Apex controller and a Visualforce page in an org that displays records with a custom filter consisting of a combination of picklist values selected by the user. The page takes too long to display results for some of the input combinations, while for other input choices it throws the exception, "Maximum view state size limit exceeded". What step should the developer take to resolve this issue?

A. Adjust any code that filters by picklist values since they are not indexed.

B. Remove instances of the transient keyword from the Apex controller to avoid the view state error.

C. Split the layout to filter records in one Visualforce page and display the list of records in a second page using the same Apex controller.

D. Use a StandardSetController or SOQL LIMIT in the Apex controller to limit the number of records displayed at a time.

Answer: (SHOW ANSWER)

The two symptoms described-slow load times and "Maximum view state size limit exceeded"-are both caused by Large Data Volumes (LDV) being handled improperly in the UI. When a user selects a filter that returns thousands of records, the SOQL query takes longer to execute, and the resulting list is stored in the controller's memory. Because Visualforce serializes all non-transient controller variables into a hidden form field (the View State), large lists quickly exceed the 135KB limit.

The correct resolution is to implement Pagination (Option D). By using the StandardSetController, a developer can easily manage large sets of data by loading only a small subset (e.g., 20 records) into the View State at a time. Users can then navigate through "Pages" of data. Alternatively, adding a LIMIT to the SOQL query prevents the application from attempting to process more data than the platform limits allow.123 Option A is incorrect because picklist fields can be indexed by Salesforce Support or if they are part of a custom index. Option B is the opposite of what is neede4d; adding transient reduces the View State. Option C adds unnecessary complexity without addressing the underlying data volume issue. Using a StandardSetController provides a built-5in, effi6cient way to handle large result sets while keeping the View State small and the page responsive.

NEW QUESTION: 22

A developer is tasked with creating a Lightning web component that is responsive on various devices. Which two components should help accomplish this goal?

A. Lightning-input-location

B. Lightning-layout

C. Lightning-navigation

D. Lightning-layout-item

Answer: (SHOW ANSWER)

To build a responsive user interface in Lightning Web Components (LWC), developers rely on the layout system provided by the Lightning Design System (SLDS). This system is

primarily implemented through two tightly coupled components: lightning-layout and lightning-layout-item.
lightning-layout acts as a flexible container (a flexbox wrapper). It allows developers to arrange child components in a row or column and manage horizontal and vertical alignment. It provides the overall structure for the grid. lightning-layout-item is the child component that resides within a lightning-layout. It is responsible for defining the actual proportions of the content across different breakpoints (small, medium, and large devices) using attributes like size, small-device-size, medium-device-size, and large-device-size. Together, these two components allow a developer to specify exactly how much space a piece of content should occupy depending on the user's screen size. For example, a sidebar might occupy 100% of the width on a phone but only 25% on a desktop. lightning-input-location is a specific input field for geolocation data, and lightning-navigation is a service used for page routing; neither is involved in the visual responsiveness or structural layout of the component. Therefore, options B and D are the essential building blocks for responsive design.

NEW QUESTION: 23
A developer is writing code that requires making callouts to an external web service. Which scenario necessitates that the callout be made in an asynchronous method?
A. The callout could take longer than 60 seconds to complete.
B. The callouts will be made in an Apex trigger.
C. Over 10 callouts will be made in a single transaction.
D. The callouts will be made using the REST API.
Answer: B (LEAVE A REPLY)
In Salesforce, a critical restriction is that synchronous callouts cannot be made from within an Apex trigger.
This architectural limitation is in place because triggers execute as part of a database transaction. Allowing a synchronous callout-which waits for a response from an external system-would keep the database connection and transaction locks open for an indeterminate amount of time. This could lead to severe performance bottlenecks and row-locking issues across the entire organization. Therefore, if a developer needs to initiate an external integration based on a record change (trigger), the logic must be handed off to an asynchronous process, such as a @future(callout=true) method, a Queueable class, or a Platform Event.
Regarding the other options: A callout that takes longer than the maximum timeout (120 seconds) will fail regardless of whether it is synchronous or asynchronous, though async is often preferred for longer-running tasks to avoid blocking the user UI. Salesforce allows up to 100 callouts in a single transaction, so making more than 10 (Option C) does not inherently force an asynchronous approach unless the 100-callout limit is exceeded. Finally, the use of the REST API (Option D) is simply a protocol choice and does not dictate the execution context. Consequently, the presence of an Apex trigger is the only

scenario listed that programmatically mandates the use of asynchronous execution to perform a callout.

NEW QUESTION: 24
A company's support process dictates that any time a case is closed with a status of 'Could not fix,' an Engineering Review custom object record should be created and populated with information from the case, the contact, and any of the products associated with the case. What is the correct way to automate this using an Apex trigger?12
A. A before update trigger on Case that creates the Engineering Review record and inserts it3456
B. An after upset trigger on Case that creates the Engineering Review record and ins7erts it8910
C. 11 A before upset trigger on Case that creates the Engineering Review record and inserts it1213
D. An after update trigger14 on Ca15se that creates the Engineering Review record and inserts it
Answer: D (LEAVE A REPLY)
In Salesforce Apex triggers, choosing the correct context (before vs. after) is critical for performance and data integrity. When the business requirement involves creating or modifying related records (records other than the one that fired the trigger), an after trigger is the platform-standard choice.
An after update trigger (Option D) is ideal here because it ensures that the Case record has been successfully validated and saved to the database (obtaining a valid timestamp and state) before the child Engineering_Review__c record is generated. Furthermore, an after trigger provides access to the read-only Trigger.new and Trigger.old maps, which contain the final field values after system validation rules and flows have run. Performing DML operations to insert a new object within a before trigger is technically possible but discouraged, as it can lead to issues if the primary record fails a subsequent system validation, potentially leaving orphaned data or causing complicated rollback scenarios. Options B and C are incorrect because "upsert" is not a valid trigger event; the valid events are insert, update, delete, and undelete. Option A is incorrect because before triggers should primarily be used for updating fields on the same record that initiated the trigger to avoid redundant DML calls. Therefore, an after update trigger provides the most stable and scalable context for cross-object record creation.

NEW QUESTION: 25
A query using OR between a Date and a RecordType is performing poorly in a Large Data Volume environment. How can the developer optimize this?
A. Break down the query into two individual queries and join the two result sets.
B. Annotate the method with the @Future annotation.
C. Use the Database.querySelector method to retrieve the accounts.

D. Create a formula field to combine the CreatedDate and RecordType value, then filter based on the formula.

Answer: A (LEAVE A REPLY)

Comprehensive and Detailed Explanation:

In SOQL, using the OR operator often prevents the Query Optimizer from using indexes effectively, especially if the fields involve different types of data. This is known as a "non-selective" query structure in LDV environments.

By breaking the query into two separate SOQL statements (Option A), the developer allows each query to be evaluated independently.

* SELECT ... FROM Account WHERE CreatedDate = :thisDate (Uses the standard index on CreatedDate).
* SELECT ... FROM Account WHERE RecordTypeId = :goldenRT (Uses the standard index on RecordTypeId).

The developer can then combine the results into a Map<Id, Account> to ensure uniqueness (preventing duplicates if a record meets both criteria). This approach ensures both queries are "selective" and run much faster than a single query with an OR filter.

Option D is a common anti-pattern because formulas are generally not indexed unless they are "deterministic" and a custom index is requested from Salesforce support; even then, filtering on formulas is often slower than direct field filters. Option B and C do not address the underlying database performance issue.

NEW QUESTION: 26

Universal Containers has an Apex trigger on Account that creates an Account Plan record when an Account is marked as a Customer. Recently a record-triggered flow was added to update a date field. Since the addition of the flow, two Account Plan records are created.

What might cause this to happen?

A. The Apex trigger is not bulk safe and calls insert inside of a for loop.

B. The flow is configured to use an 'Update Records' element.

C. The flow is configured to evaluate when a record is created and every time it is edited.

D. The Apex trigger does not use a static variable to ensure it only fires once.

Answer: D (LEAVE A REPLY)

This is a classic case of trigger recursion caused by the Salesforce Order of Execution.

When an Account is updated to "Customer," the following occurs:

* The original "Before" and "After" triggers run. The After trigger creates the first Account Plan.
* The Record-Triggered Flow executes and updates the "Customer Since" date field.
* Because a Flow (or Workflow) performed a field update, Salesforce re-triggers the Apex triggers on the Account object to ensure any logic dependent on that new date field is executed.
* During this second pass, the After trigger runs again and, seeing the status is still "Customer," creates a second Account Plan.

To prevent this, developers use a static variable (Option D) in a handler class to act as a "recursion guard." Static variables persist for the duration of the entire transaction. By checking this variable at the start of the trigger, the code can detect that it has already run once and skip the record creation on the second pass. Option B and C describe standard Flow behavior but are not the "fix" for the trigger logic. Option A would cause issues with limits but wouldn't necessarily cause a double-fire unless recursion was involved.

NEW QUESTION: 27
A developer needs to add code to a Lightning web component's configuration file so the component only renders for a desktop size form factor when on a record page. What should the developer add to the component's record page target configuration to meet this requirement?
A. <supportedFormFactors> <supportedFormFactor type="Large"/>
</supportedFormFactors>
B. <lightningLayout> </lightningLayout>
C. <design> ...
D. <property name="formFactor" value="Large"> ...
Answer: A (LEAVE A REPLY)
In a Lightning Web Component's configuration file (.js-meta.xml), the <supportedFormFactors> tag allows the developer to specify which device form factors the component supports. To restrict the component to only render on desktop devices, the developer must specify the type="Large".
The structure is:
XML
<targetConfigs>
<targetConfig targets="lightning__RecordPage">
<supportedFormFactors>
<supportedFormFactor type="Large"/>
</supportedFormFactors>
</targetConfig>
</targetConfigs>
If supportedFormFactors is not specified, the component defaults to supporting both Small (phone) and Large (desktop) form factors. Options B, C, and D are incorrect syntax for defining form factor support in LWC metadata.

NEW QUESTION: 28
A developer built an Aura component for guests to self-register upon arrival at a front desk kiosk. Now the developer needs to create a component for the utility tray to alert users whenever a guest arrives at the front desk. What should be used?
A. DML Operation
B. ChangeLog

C. Application Event
D. Component Event16
Answer: C (LEAVE A REPLY)
18
In the Aura Component framework, communication between components is handled via events. When two components need to communicate but do19 not share a direct parent-child relationship-such as a kiosk registration component on a page and an alert component residing in the global utility tray-an Application Event (Option C) is required. Application events follow a "publish-subscribe" model. One component fires the event, and any other component in the application that has a handler for that specific event type can listen for and react to it, regardless of where they sit in the component hierarchy. This is distinct from a Component Event (Option D), which is restricted to bubbling up the component containment hierarchy (from child to parent).
While modern implementations might also consider the Lightning Message Service (LMS) for cross- framework communication (between Aura, LWC, and Visualforce), within the specific context of Aura-to- Aura communication for decoupled components like utility bar items, Application Events remain the fundamental architectural solution. DML Operations (Option A) and ChangeLogs (Option B) are database- level concepts and do not provide the real-time, client-side notification capabilities needed for a UI alert system.

NEW QUESTION: 29
Which tag should a developer use to display different text while an <apex:commandButton> is processing an action?
A. <apex:actionStatus>
B. <apex:actionPoller>
C. <apex:pageMessages>
D. <apex:actionSupport>
Answer: A (LEAVE A REPLY)
In Visualforce, the <apex:actionStatus> component is specifically designed to provide user feedback during asynchronous (AJAX) operations. When an <apex:commandButton> or <apex:actionSupport> initiates a request to the server, there is a delay while the server processes the logic and returns the updated data. To prevent users from clicking the button multiple times or to simply inform them that the system is working, developers use <apex:actionStatus>.
This tag allows you to define two facets: startText (or a start facet) and stopText (or a stop facet). By linking the status attribute of the <apex:commandButton> to the id of the <apex:actionStatus>, the page will automatically toggle between these states. For example, while the button is processing, the status can display
"Processing..." or a loading spinner, and once the action is complete, it can return to being hidden or display a
"Completed" message.

Option B is used for timed, repeated requests. Option C is for displaying validation or system errors. Option D is used to add AJAX support to other components (like a picklist) but doesn't manage the display state of the request itself. Therefore, <apex:actionStatus> is the standard tool for managing UI state during the request- response lifecycle in Visualforce.

NEW QUESTION: 30
A developer created a JavaScript library that simplifies the development of repetitive tasks and features and uploaded the library as a static resource called jsUtils in Salesforce. Another developer is coding a new Lightning web component (LWC) and wants to leverage the library. Which statement properly loads the static resource within the LWC?
A. import jsUtilities from '@salesforce/resourceUrl/jsUtils';
B. import {jsUtilities} from '@salesforce/resourceUrl/jsUtils';
C. const jsUtility = $A.get('$Resource.jsUtils');
D. <lightning=require scripts="{$Resource.jsUtils}"/>
Answer: (SHOW ANSWER)
Comprehensive and Detailed 150 to 25042 words o43f Explanation:
In Lightning Web Components, external JavaScript libraries or CSS files stored in Static Resources must be imported using a specific syntax. The correct way to reference the resource URL is through the @salesforce
/resourceUrl scoped module. Option A correctly follows this convention: import resourceName from
'@salesforce/resourceUrl/staticResourceName'. This statement provides the component with the URL string pointing to the resource.
It is important to note that the import statement only gives you the URL of the file. To actually load and execute the JavaScript within the component, you would then need to use the loadScript function from the lightning/platformResourceLoader module, typically within the renderedCallback() lifecycle hook.
Option B is incorrect because resource imports are default imports, not named imports. Option C uses Aura framework syntax ($A.get), which is not supported in LWC. Option D uses a syntax that resembles Aura's ltng:require tag, which does not exist in the LWC template syntax. Therefore, Option A is the essential first step for any LWC to identify and eventually load a static resource.

NEW QUESTION: 31
Universal Containers uses a custom Lightning page to provide a mechanism to perform a step-by-step wizard search for Accounts. One of the steps in the wizard is to allow the user to input text into a text field, ERP_Number__c, that is then used in a query to find matching Accounts.
Java
erpNumber = erpNumber + '%';

List<Account> accounts = [SELECT Id, Name FROM Account WHERE ERP_Number__c LIKE :
erpNumber];
A developer receives the exception 'SOQL query not selective enough'. Which step should be taken to resolve the issue?
A. Move the SOQL query to within an asynchronous process.
B. Mark the ERP_Number__c field as required.
C. Mark the ERP_Number__c field as an external ID.
D. Change the query to use a SOSL statement instead of SOQL.
Answer: (SHOW ANSWER)
The "SOQL query not selective enough" error occurs when a query on an object with a large volume of records (typically over 200,000) does not use a filtered index to narrow down the result set effectively. In Sales1force,2 the query optimizer must be able to use an index to scan a small enough percentage of the total records to maintain performance. When a query filters on a non-indexed field, the system is forced to perform a full table scan, which triggers this exception to prevent performance degradation.
To resolve this, the developer must ensure that the field used in the WHERE clause is indexed. Marking a custom field as an "External ID" or "Unique" automatically creates a custom index on that field. By marking ERP_Number__c as an External ID, the platform generates a underlying database index, allowing the query optimizer to quickly locate matching records without scanning the entire table. While the LIKE operator with a trailing wildcard (e.g., 'Text%') can utilize an index, it will only be considered "selective" if the filter narrow the results below the platform's selectivity thresholds (typically 10% of the first million records). Making the field an External ID is the standard programmatic solution to provide the necessary indexing for selectivity.
Option A is incorrect because moving a non-selective query to an asynchronous process does not change the fact that it is non-selective; it will still fail.

NEW QUESTION: 32
An Apex trigger creates a Contract record every time an Opportunity is marked as Closed/Won. Historical records need to be loaded, but Contract records should not be

created during this mass load. What is the most extendable way to update the Apex trigger to accomplish this?
A. Add the Profile ID of the user who loads the data to the trigger.
B. Add a validation rule to the Contract to prevent creation by the data load user.
C. Use a list custom setting to disable the trigger for the user who loads the data.
D. Use a hierarchy custom setting to skip executing the logic inside the trigger for the user who loads the data.
Answer: D (LEAVE A REPLY)
When managing trigger logic that needs to be conditionally bypassed (often called a "Trigger Kill Switch"), using Hierarchy Custom Settings (Option D) is the best practice for scalability and maintainability.
A developer can create a Hierarchy Custom Setting named Trigger_Settings__c with a checkbox field Disable_Opportunity_Trigger__c. In the Apex trigger, the code should first check this setting:
if (Trigger_Settings__c.getInstance().Disable_Opportunity_Trigger__c) return; Because it is a hierarchy setting, an administrator can enable this checkbox specifically for the User or Profile performing the data load without affecting the daily operations of other sales reps. Once the data load is complete, the setting can be toggled off. This approach is "extendable" because it doesn't require hardcoding IDs (Option A), which change between environments (Sandbox vs. Production).
Option C (List Custom Settings) is less flexible because it doesn't support the Profile-to-User hierarchy.
Option B (Validation Rules) would stop the record creation but would result in a DML error in the trigger, potentially failing the entire data load transaction rather than simply skipping the logic. Hierarchy Custom Settings provide a clean, metadata-driven way to control execution flow across different user contexts.

NEW QUESTION: 33
A company has a Lightning page with many Lightning Components, some that cache reference data. It is reported that the page does not always show the most current reference data. What can a developer use to analyze and diagnose the problem in the Lightning page?
A. Salesforce Lightning Inspector Storage tab12
B. Salesforce Lightning Inspector Event Log tab34
C. Salesforce Lightning Inspector Actions tab56
D. Salesforce Lightning Inspector Transactions tab78
Answer: (SHOW ANSWER)
Comprehensive and Detailed 150 to 25110 words of Explana12tion:
The Salesforce Lightning Inspector is a Chrome DevTools extension used to investigate and optimize Lightning components. When a page fails to show the most current reference data, the issue often lies within the client-side caching mechanism. The Storage tab (A) is

specifically designed to help developers inspect the client-side data stores used by the Lightning Data Service (LDS) and custom component caching.
In this tab, a developer can view the contents of the various storage caches (such as actions, records, and lists). It allows the developer to see exactly what data is currently stored in the cache, its expiration time, and whether the data is being retrieved from the server or served directly from the cache. By analyzing the Storage tab, a developer can determine if a component is incorrectly holding onto "stale" data instead of refreshing from the server.
The Actions tab (C) is useful for monitoring the status of server-side controller actions, and the Event Log (B) tracks the sequence of application and component events. However, for a data consistency issue specifically related to caching, the Storage tab provides the most direct visibility into the underlying cause. Using this tool, a developer can verify if the cache invalidation logic is working correctly or if the storage limits are causing unexpected data eviction.

NEW QUESTION: 34
A developer is writing a Jest test for a Lightning web component that conditionally displays child components based on a user's checkbox selections. What should the developer do to properly test that the correct components display and hide for each scenario?
A. Add a teardown block to reset the DOM after each test.9
B. Create a new describe block for each test.10
C. Create a new j.sdom instance for each test.11
D. Reset the DOM after each test with the afterEach() method.12
Answer: D (LEAVE A REPLY)
Comprehensive and Detailed 150 to 14250 words of Explanation:
Jest tests for Lightning Web Components (LWC) run in a virtual browser environment called JSDOM. In this environment, the document.body persists across individual test cases (it or test blocks) within a single test file.
If a developer mounts a component to the DOM in the first test and doesn't remove it, the component's state and DOM elements will still be present when the second test starts, leading to "leaked" data and unreliable test results.
To ensure test isolation-which is critical for testing conditional rendering-the developer must clean up the DOM after every test. The standard best practice is to use the afterEach() hook (Option D). Within this hook, the developer should execute a loop that removes all child elements from document.body. This ensures that each test begins with a completely "clean slate." Option A is too vague; "teardown block" is a general term, but afterEach() is the specific Jest implementation.
Option B is for organizing tests but doesn't handle DOM cleanup. Option C is unnecessary as the LWC testing utility manages the JSDOM instance. By using afterEach() to reset the DOM, the developer can accurately verify that the component correctly shows or hides child components based on the specific inputs provided in each individual test case.

NEW QUESTION: 35
The test method tests an Apex trigger that the developer knows will make a lot of queries when a lot of Accounts are simultaneously updated. The test method fails at Line 20 because of "too many SOQL queries." What is the correct way to fix this?
Java
Line 12: //Set accounts to be customers
Line 13: for(Account a : DataFactory.accounts)
Line 14: {
Line 15: a.Is_Customer__c = true;
Line 16: }
Line 17:
Line 18: update DataFactory.accounts;
Line 19:
Line 20: List<Account> acctsAfter = [SELECT Number_Of_Transfers__c FROM Account WHERE Id IN :
DataFactory.accounts];
A. Use Limits.getLimitQueries() to find the total number of queries that can be issued.
B. Change the DataFactory class to create fewer Accounts so that the number of queries in the trigger is reduced.
C. Add Test.startTest() before and add Test.stopTest() after both Line 7 and Line 20 of the code.
D. Add Test.startTest() before and add Test.stopTest() after Line 18 of the code.
Answer: D (LEAVE A REPLY)
In Salesforce unit testing, the platform provides a mechanism to reset governor limits specifically for the code being tested. The "Too many SOQL queries" error at Line 20 occurs because the SOQL queries consumed during the Data Setup (Lines 4-10) and the Trigger Execution (Line 18) are being counted against the same transaction limit.
The correct solution is to wrap the DML operation (the code that fires the trigger) in Test.startTest() and Test.stopTest() (Option D).
* Test.startTest(): When this method is called, the code enters a new execution context with a fresh set of governor limits. The queries used to set up the data in Lines 5 and 7 will no longer count against the limit available for the update on Line 18.
* Test.stopTest(): This ensures that any asynchronous processes (like @future or Queueable) spawned by the trigger are forced to complete before the code proceeds to Line 20.
Option B is a "workaround" that reduces the validity of the test, as you should test your code against bulk volumes. Option A is for monitoring but doesn't solve the limit issue. Option C is logically incorrect because placing stopTest after Line 20 would mean the query on Line 20 is still sharing limits with the code it is trying to verify. Placing the block strictly around the operation being tested (Line 18) is the standard best practice.

NEW QUESTION: 36
A business currently has a labor-intensive process to manually upload orders from an external OMS.
Accounts must be exported to get IDs to link the orders. Which two recommendations should make this process more efficient?
A. Use the insert wizard in the Data Loader to import the data.
B. Ensure the data in the file is sorted by the order ID.
C. Use the upsert wizard in the Data Loader to import the data.
D. Identify unique fields on Order and Account and set them as External IDs.
Answer: C,D (LEAVE A REPLY)
The current bottleneck is caused by the need to resolve Salesforce Internal IDs ($001...$) for the Account lookup before orders can be imported. This can be completely bypassed by using External IDs (Option D) and the Upsert operation (Option C).
* External IDs: By marking a unique field from the OMS (like OMS_Account_Number__c) as an External ID in Salesforce, the developer creates a "foreign key" that Salesforce understands.
* Upsert Wizard: When using the Data Loader's Upsert wizard, the user is asked which field should be used to match the parent Account. By selecting the OMS_Account_Number__c external ID, the Data Loader allows the user to map the orders directly to accounts using the OMS number found in the CSV file.
Salesforce will automatically find the correct Account ID in the background. This eliminates the "export-and- VLOOKUP" step entirely. Option A (Insert) would still require internal IDs. Option B (sorting) helps with performance in massive loads but doesn't solve the manual ID matching problem. Using Upsert with External IDs is the platform-recommended way to streamline recurring data integrations.

NEW QUESTION: 37
A developer creates an application event that has triggered an infinite loop. What may have caused this problem?
A. The event has multiple handlers registered in the project.
B. The event is fired from a custom renderer.
C. An event is fired 'ontouched' and is unhandled.
D. The event handler calls a trigger.
Answer: (SHOW ANSWER)
In the Aura Component Framework, an infinite loop involving an application event is frequently caused by firing the event from a custom renderer (Option B), specifically from the afterRender or rerender functions.
The framework's rendering lifecycle is sensitive. When a component's state changes, the renderer is invoked to update the DOM. If a developer places code inside a renderer that fires an application event, and that event-or a handler of that event-subsequently modifies

an attribute on the same component, the component is marked as "dirty." This triggers the renderer to run again, which fires the event again, creating an infinite loop.
Option A is incorrect because multiple handlers simply result in multiple independent blocks of code executing, not a loop. Option C relates to touch events on mobile but wouldn't cause a loop unless the handler itself re-triggered the event. Option D is incorrect because triggers are server-side and do not directly re-fire client-side application events in a looping fashion. To avoid this, event firing should be restricted to controller or helper actions that are triggered by specific user interactions or discrete system states, rather than the automatic rendering cycle.

NEW QUESTION: 38
The test method calls an @future method that increments a value. The assertion is failing because the value equals 0. What is the optimal way to fix this?
Java
@isTest
static void testIncrement() {
Account acct = new Account(Name = 'Test', Number_Of_Times_Viewed__c = 0); insert acct; AuditUtil.incrementViewed(acct.Id); // This is the @future method Account acctAfter = [SELECT Number_Of_Times_Viewed__c FROM Account WHERE Id = :acct.Id][0]; System.assertEquals(1, acctAfter.Number_Of_Times_Viewed__c);
}
A. Change the assertion to System.assertEquals(0, acctAfter.Number_Of_Times_Viewed__c).
B. Add Test.startTest() before and Test.stopTest() after insert acct.
C. Change the initialization to acct.Number_Of_Times_Viewed__c = 1.
D. Add Test.startTest() before and Test.stopTest() after AuditUtil.incrementViewed.
Answer: D (LEAVE A REPLY)
Asynchronous methods, such as those annotated with @future, do not run immediately when called in Apex.
Instead, they are added to a queue to be processed when system resources become available. In a unit test, if you call a future method and immediately query the database for the result, the future method likely hasn't executed yet, resulting in the "0" value observed in the assertion.
To test asynchronous code, you must wrap the call within Test.startTest() and Test.stopTest() (Option D).
When the code reaches Test.stopTest(), the execution of the test script pauses, and the system forces all queued asynchronous jobs (future methods, batch jobs, queueable jobs) to run to completion synchronously.
By placing the AuditUtil.incrementViewed(acct.Id) call inside this block, you ensure that by the time the next line of code (the SOQL query) runs, the increment logic has finished and the database reflects the new value.

Option B is incorrect because inserting the account is a synchronous operation that doesn't require the stopTest wait. Option A and C avoid the problem rather than testing the logic correctly. Option D is the standard platform-required pattern for testing asynchronous side effects.

NEW QUESTION: 39
A Salesforce org has more than 50,000 contacts. A new business process requires a calculation that aggregates data from all of these contact records. This calculation needs to run once a day after business hours. Which two steps should a developer take to accomplish this?
A. Use the @future annotation.
B. Implement the Database.Batchable interface.
C. Implement the Schedulable interface.
D. Implement the Queueable interface.
Answer: B,C (LEAVE A REPLY)
This scenario presents two distinct requirements: processing a large volume of data (50,000 records) and running the process at a specific time (once a day after business hours).
* Database.Batchable (Option B): Processing 50,000 records in a single synchronous transaction will exceed Apex Governor Limits (specifically the CPU time limit or the heap size limit). Batch Apex allows the system to process these records in smaller chunks (batches), ensuring that the calculation completes without hitting limits.
* Schedulable (Option C): To ensure the job runs "once a day after business hours," the Schedulable interface is required. The developer allows the class to be scheduled via the Apex Scheduler or the UI.
The standard pattern for this use case is to create a class that implements Schedulable, and inside its execute method, instantiate and execute the Batch class (Database.executeBatch). This combines precise timing with bulk processing capabilities.

NEW QUESTION: 40
Universal Containers wants to develop a recruiting app for iOS and Android via the standard Salesforce mobile app. It has a custom user interface design and offline access is not required. What is the recommended approach to develop the app?
A. Salesforce SDK
B. Lightning Web Components
C. Lightning Experience Builder
D. Visualforce
Answer: B (LEAVE A REPLY)
When developing custom functionality to be surfaced within the standard Salesforce Mobile App, Lightning Web Components (LWC) (Option B) is the recommended approach.

LWCs are the modern, high- performance standard for Salesforce development, utilizing modern web standards (ES6+, Web Components) that run natively in the browser.
LWC is preferred over the other options for several reasons:
* Performance: LWCs are lightweight and provide a much faster, more responsive user interface than Visualforce (Option D), which is critical for mobile user experience.
* Standardization: Since the app will be accessed via the standard Salesforce mobile app, LWCs integrate seamlessly into the mobile navigation, tabs, and action menus.
* Customization: LWCs provide full control over the CSS and HTML, allowing the developer to meet the "custom user interface design" requirement while still leveraging the Lightning Data Service for efficient data handling.
Option A (Salesforce SDK) is for building "Custom/Native" standalone apps, not for extending the standard Salesforce app. Option C (Experience Builder) is primarily for external sites (Communities). LWC provides the perfect balance of "custom design" and "native integration" for internal mobile users.

NEW QUESTION: 41
A developer created a class that implements the Queueable Interface, as follows:
Java
public class without sharing OrderQueueableJob implements Queueable {
public void execute(QueueableContext context) {
// implementation logic
System.enqueueJob(new FollowUpJob());
}
}
As part of the deployment process, the developer is asked to create a corresponding test class. Which two actions should the developer take to successfully execute the test class?
1
A. Implement seeAllData=true to ensure the Queueable job is able to run in bulk mode.2
B. Implement Te3st.isRunningTest() to prevent chaining jobs during test execution.
C. Ensure the running user of the test class has, at least, the View All permission on the Order object.
D. Enclose System.enqueueJob(new OrderQueueableJob()) within Test.startTest and Test.stopTest().
Answer: (SHOW ANSWER)
Testing asynchronous Apex, such as the Queueable interface, requires specific handling to ensure the job actually executes during the test run and adheres to platform limits.
First, Salesforce enforces a strict limit on chaining Queueable jobs during unit tests. Specifically, you cannot chain jobs (enqueue a job from another job) within a test. Since the OrderQueueableJob attempts to enqueue FollowUpJob, the test will throw an error. To resolve this, the developer should use Test.isRunningTest() (Option B) within the execute method to conditionally bypass the System.enqueueJob call during test runs.

Second, asynchronous jobs are queued by the system and do not run immediately. To force the execution of a Queueable job so that results can be asserted, the System.enqueueJob call must be wrapped within Test.
startTest() and Test.stopTest() (Option D). When Test.stopTest() is called, the execution pauses until all queued asynchronous jobs in that block finish running.
Option A is incorrect because seeAllData does not affect asynchronous processing modes.
Option C is incorrect because the class is defined as without sharing, meaning it bypasses sharing rules regardless of the running user's permissions. By combining recursion/chaining guards and the startTest/stopTest block, the developer can reliably test the logic within the execute method.

NEW QUESTION: 42
A developer is building a Lightning web component that retrieves data from Salesforce and assigns it to the record property:
JavaScript
import { LightningElement, api, wire } from 'lwc';
import { getRecord } from 'lightning/uiRecordApi';
export default class Record extends LightningElement {
@api fields;
@api recordId;
record;
}
What must be done in the component to get the data from Salesforce?
A. Add @api(getRecord, { recordId: '$recordId' }) above record.
B. Add @wire(getRecord, { recordId: '$recordId' }) above record.
C. Add @api(getRecord, { recordId: '$recordId', fields: '$fields' }) above record.
D. Add @wire(getRecord, { recordId: '$recordId', fields: '$fields' }) above record.
Answer: D (LEAVE A REPLY)
To retrieve record data in a Lightning Web Component using the Lightning Data Service (LDS), the developer must use the @wire decorator with the getRecord wire adapter.
Option D provides the correct syntax. The @wire decorator requires the adapter name (getRecord) and a configuration object. The configuration must include the recordId and either the fields or layoutTypes to be retrieved. By using the $ prefix (e.g., '$recordId'), the developer makes the property "reactive." This means that whenever the value of recordId or fields changes, the wire service automatically re-provisions the data from Salesforce.
Options A and C are incorrect because @api is used to expose public properties, not for wiring data. Option B is incorrect because getRecord requires a field list or layout type to know which data points to fetch from the server. Using the reactive $fields ensures that the component remains flexible and can load different field sets as defined by its parent component.

NEW QUESTION: 43
A developer is trying to access org data from within a test class. Which sObject type requires the test class to have the (seeAllData=true) annotation?
A. RecordType
B. Report
C. User
D. Profile
Answer: B (LEAVE A REPLY)
Comprehensive and Detailed Explanation:
Salesforce enforces test isolation, meaning unit tests do not have access to the data in the organization by default. However, some objects are considered "metadata-like" or "setup objects" and are always visible to tests without special annotations. These include Users (Option C), Profiles (Option D), and RecordTypes (Option A).
Conversely, some objects are considered "data" but cannot be easily created within a test method due to their complexity or nature. Reports (Option B), along with Pricebooks and Folders, fall into a category where access to existing org data is often required because the platform does not support DML operations to create them in a test context. To query for an existing Report record in a unit test, the test class or method must be annotated with @isTest(seeAllData=true).
Note: It is a best practice to avoid seeAllData=true whenever possible to ensure tests are deterministic and independent of the environment, but it remains a requirement for Report-related logic.

NEW QUESTION: 44
A developer needs to implement a system audit feature that allows users, assigned to a custom profile named
"Auditors", to perform searches against the historical records in the Account object. The developer must ensure the search is able to return history records that are between 6 and 12 months old. Given the code below, which select statement should be inserted as a valid way to retrieve the Account History records?4445 Java Date initialDate = System.Today().addMonths(-12); Date endDate = System.Today().addMonths(-6);
// Insert SELECT statement here
A. [SELECT AccountId, CreatedDate, Field, NewValue, OldValue FROM Account_History WHERE CreatedDate <= :initialDate AND CreatedDate <= :endDate];
B. [SELECT AccountId, CreatedDate, Field, NewValue, OldValue FROM Account_History WHERE CreatedDate >= :endDate AND CreatedDate <= :initialDate];
C. [SELECT AccountId, CreatedDate, Field, NewValue, OldValue FROM AccountHistory WHERE CreatedDate BETWEEN :initialDate AND :endDate];
D. [SELECT AccountId, CreatedDate, Field, NewValue, OldValue FROM AccountHistory WHERE CreatedDate => :initialDate AND CreatedDate <= :endDate];
Answer: C (LEAVE A REPLY)

To query history records for a standard object like Account, the developer must use the correct API name, which is AccountHistory (not Account_History). When filtering for a range of dates, SOQL provides the BETWEEN operator, which makes the query cleaner and more readable. Option C uses both the correct object name and the BETWEEN syntax, making it the most efficient solution.
In SOQL, CreatedDate BETWEEN :initialDate AND :endDate is shorthand for CreatedDate >= :initialDate AND CreatedDate <= :endDate. In the provided code, initialDate is 12 months ago (the older date) and endDate is 6 months ago (the more recent date). Therefore, the range correctly captures records between those two timestamps.
Option A and B use an incorrect object name (Account_History). Option D uses the incorrect comparison operator => (the correct operator is >=). Furthermore, Option C follows the best practice for date range queries in Apex by using bind variables with the standard AccountHistory object, ensuring the "Auditors" profile can retrieve the necessary historical data within the platform's audit and security constraints.

NEW QUESTION: 45
Salesforce users consistently receive a "Maximum trigger depth exceeded" error when saving an Account.
How can a developer fix this error?
A. Split the trigger logic into two separate triggers.
B. Modify the trigger to use the isMultiThread=true annotation.
C. Convert the trigger to use the @future annotation, and chain any subsequent trigger invocations to the Account object.
D. Use a helper class to set a Boolean to TRUE the first time a trigger is fired, and then modify the trigger to only fire when the Boolean is FALSE.
Answer: D (LEAVE A REPLY)
1
The "Maximum trigger depth exceeded" error occurs when a recursive loop is created, causing triggers to fire repeatedly until the platform's limit of 16 recursive cal2ls is reached. This often happens when an after update trigger performs a DML operation on the same record that initiated the trigger, or when two different objects have triggers that update each other in a circular fashion.
To resolve this, developers use a static Boolean variable within a helper class to manage the execution state.
Because static variables in Apex persist for the duration of a single transaction, the trigger can check the value of this Boolean before executing its logic. When the trigger runs for the first time, it checks if the Boolean is FALSE, sets it to TRUE, and then proceeds. If the trigger is re-invoked within the same transaction (recursion), the Boolean check will fail, and the logic will be skipped. This "recursion guard" ensures the logic only runs once per transaction.

Splitting the logic into two triggers (A) would not help, as both triggers would still be part of the same recursive cycle. There is no isMultiThread annotation (B), and while @future (C) can break the immediate execution chain, it does not address the underlying logic flaw and can lead to unmanageable asynchronous overhead. The static variable approach is the industry-standard "best practice" for recursion control.

NEW QUESTION: 46
Consider the following code snippet:
HTML
<apex:page docType="html-5.0" controller="FindOpportunities">
<apex:form >
<apex:pageBlock >
<apex:pageBlockSection title="find opportunity">
<apex:input label="opportunity name"/>
<apex:commandButton value="search" action="{!search}"/>
</apex:pageBlockSection>
<apex:pageBlockSection title="Opportunity List" id="opportunityList">
</apex:pageBlockSection>
</apex:pageBlock>
</apex:form>
</apex:page>
Users of this Visualforce page complain that the page does a full refresh every time the Search button is pressed. What should the developer do to ensure that a partial refresh is made so that only t13he section identified with opportunityList is re-drawn on the screen?
1415
A. Enclose the DAT16A table within the <apex:actionRegion> tag.1718
B. Implement the <apex:actionFunction> tag with immediate = true.
C. Ensure the action method search returns null.19
D. Implement the reRender attribute on the <apex:commandButton> tag.
Answer: D (LEAVE A REPLY)
In Visualforce, the standard behavior for an <apex:commandButton> is to perform a full page postback. To convert this into an asynchronous (AJAX) request that only updates a specific portion of the page, the developer must use the reRender attribute (D). By setting reRender="opportunityList" on the command button, the developer instructs the platform to capture the server's response and update only the DOM element with that specific ID, rather than refreshing the entire browser window.
While Option C (returning null from the search method) is a requirement for staying on the same page, it does not prevent the full refresh by itself. Option A (<apex:actionRegion>) is used to limit which part of the form's data is sent to the server for processing (to avoid validation errors on unrelated fields), but it does not control which part of the page is re-drawn on the client. Option B (<apex:actionFunction>) is a way to call Apex from

JavaScript but doesn't solve the button's refresh behavior directly. Implementing reRender is the standard and most efficient way to achieve the "partial refresh" UX in Visualforce, significantly improving the perceived performance and user experience by reducing the amount of data and UI flickering during search operations.

NEW QUESTION: 47
The use of the transient keyword in Visualforce page helps with which performance issue?
A. Reduces view state
B. Improves query performance
C. Reduces load times
D. Improves page transfers
Answer: (SHOW ANSWER)
In Visualforce, the "View State" is a hidden form field that maintains the state of the page (and the controller's variables) across postbacks to the server. If the View State becomes too large, it can slow down page loads and eventually hit the Salesforce View State limit (170KB), causing the page to crash.
The transient keyword is used to declare instance variables in Apex controllers that should not be saved in the View State. When a variable is marked as transient, its value is discarded after the request finishes and is not transmitted back to the client. This effectively reduces the size of the View State. It is commonly used for data that is needed only for the duration of the current request (like a large list of records displayed in a read-only table) and can be easily requeried or recalculated if needed.

NEW QUESTION: 48
A developer is trying to decide between creating a Visualforce component or a Lightning component. Which scenario necessitates the use of Visualforce?
A. Does the screen need to be rendered as a PDF without using a third-party application?
B. (Option Not Provided in Context)
C. Will the screen make use of a JavaScript framework?
D. (Matches Option A in context of question logic)
Answer: A (LEAVE A REPLY)

While Salesforce highly recommends Lightning Web Components (LWC) or Aura components for modern UI development, there are specific legacy use cases where Visualforce is still the required technology. The most prominent of these is rendering a page as a PDF.
Visualforce provides the renderAs="pdf" attribute on the <apex:page> tag, which utilizes a server-side engine to convert the HTML markup into a downloadable PDF document. Neither Lightning Web Components nor Aura components have a native, built-in capability to render directly to PDF on the server side; they rely on client-side rendering or third-party libraries/services. Therefore, if the requirement is native PDF generation, Visualforce is the correct choice.

NEW QUESTION: 49
Which code snippet represents the optimal Apex trigger logic for assigning a Lead's Region based on its PostalCode, using a custom Region__c object?
A. Java
Set<String> zips = new Set<String>();
for(Lead l : Trigger.new) {
if(l.PostalCode != Null) {
zips.add(l.PostalCode);
}
}
List<Region__c> regions = [SELECT Zip_Code__c, Region_Name__c FROM Region__c WHERE Zip_Code__c IN :zips]; Map<String, String> zipMap = new Map<String, String>();
for(Region__c r : regions) { zipMap.put(r.Zip_Code__c, r.Region_Name__c);
}
for(Lead l : Trigger.new) {
if(l.PostalCode != Null) {
B. Region__c = zipMap.get(l.PostalCode);
}
}
C. Java
Set<String> zips = new Set<String>();
for(Lead l : Trigger.new) {
if(l.PostalCode != Null) {
zips.add(l.PostalCode);
}
}
for (Lead l : Trigger.new) {
List<Region__c> regions = [SELECT Zip_Code__c, Region_Name__c FROM Region__c WHERE Zip_Code__c IN :zips]; for (Region__c r : regions) { if(l.PostalCode == r.Zip_Code__c) {

D. Region__c = r.Region_Name__c;
}
}
}
E. Java
for (Lead l : Trigger.new) {
Region__c reg = [SELECT Region_Name__c FROM Region__c WHERE Zip_Code__c = :l.
PostalCode];
F. Region__c = reg.Region_Name__c;
}
G. Java
Set<String> zips = new Set<String>();
for(Lead l : Trigger.new) {
if(l.PostalCode != Null) {
zips.add(l.PostalCode);
}
}
for(Lead l : Trigger.new) {
List<Region__c> regions = [SELECT Zip_Code__c, Region_Name__c FROM Region__c WHERE Zip_Code__c IN :zips]; for(Region__c r : regions) { if(l.PostalCode == r.Zip_Code__c) {
H. Region__c = r.Region_Name__c;
}
}
}
Answer: A (LEAVE A REPLY)
The "optimal" Apex trigger logic is defined by its bulkification and strict adherence to Salesforce governor limits. In Apex, a trigger can process up to 200 records in a single batch. If a developer places a SOQL query inside a loop, the transaction will fail with a System.LimitException if more than 100 records are processed, as the limit is 100 synchronous SOQL queries.
Option A follows the industry-standard "Bulkify, Query, and Map" design pattern:
* Collection: It iterates through the input records once to gather all relevant search criteria (Zip Codes) into a Set.
* External Query: It performs a single SOQL query outside of any loop to fetch all matching Region__c records for the entire batch. This is the most critical step for scalability.
* Mapping: It organizes the results into a Map<String, String> (Zip Code to Region Name). Maps provide $O(1)$ constant-time lookup performance.
* Final Assignment: It iterates through the Leads a second time and assigns the Region using the Map.

Options B, C, and D are all anti-patterns. Option C is the most dangerous, as it executes a query for every single record. Options B and D are slightly better but still redundant, as they execute a query inside a loop that retrieves data already retrieved in previous iterations. Only Option A ensures the code is efficient, bulk-safe, and stays well within governor limits even under heavy data loads.

NEW QUESTION: 50
Which statement is true regarding savepoints?
A. You can roll back to any savepoint variable created in any order.
B. Static variables are not reverted during a rollback.
C. Savepoints are not limited by DML statement governor limits.
D. Reference to savepoints can cross trigger invocations.
Answer: B (LEAVE A REPLY)
In Salesforce Apex, a Database.Savepoint allows a developer to define a point in a transaction that can be returned to if an error occurs, effectively undoing database changes. However, it is critical to understand what is-and is not-affected by a rollback. According to the platform specifications, while database records (DML operations) are reverted to their state at the time the savepoint was set, static variables are not reverted (Option B). Static variables persist across the entire transaction regardless of rollbacks, which is why they are commonly used as "recursion guards" to track if a trigger has already fired.
Other options are incorrect due to the strict "stack" nature of savepoints. Savepoints must be rolled back in reverse chronological order; rolling back to an earlier savepoint invalidates any savepoints created after it (Option A). Furthermore, a savepoint variable is only valid within the specific execution context in which it was created. You cannot pass a savepoint reference across different trigger invocations or asynchronous boundaries (Option D). Finally, while the rollback itself isn't a DML statement, the operations performed between setting a savepoint and rolling back still count toward DML and CPU governor limits (Option C).
Understanding that memory state (static variables) remains unchanged during a database rollback is essential for debugging complex transactional logic.

NEW QUESTION: 51
In an organization that has multi-currency enabled, a developer is tasked with building a Lighting component that displays the top ten Opportunities most recently accessed by the logged in user. The developer must ensure the Amount and LastModifiedDate field values are displayed according to the user's locale. What is the most effective approach to ensure values displayed respect the user's locale settings?1819
A. Use the FORMAT ( ) function in the SOQL query.2021
B. Use a wrapper class to format the values retrieved via SOQL.2223
C. Use REGEX expressions to format the values retrieved via SOQL.2425

D. Use the FOR VIEW clause in the SOQL query.2627
Answer: A (LEAVE A REPLY)
Co30mprehensive and Detailed 150 to 250 words of Explanation:
When retrieving data for display in a custom UI, formatting dates, times, and currencies to match the user's specific locale can be complex. Salesforce provides the FORMAT() function within SOQL specifically to handle this at the database layer.
When you wrap a field in the FORMAT() function, such as SELECT FORMAT(Amount), FORMAT (LastModifiedDate) FROM Opportunity, the platform automatically applies the localized formatting rules based on the user's "Language and Locale" settings. This includes applying the correct currency symbol, thousands separators, decimal points, and date/time structures (e.g., MM/DD/YYYY vs DD/MM/YYYY). For organizations with Multi-Currency enabled, FORMAT() also handles the conversion and display of currency symbols according to the record's currency code.
Using FORMAT() is significantly more efficient than creating a wrapper class (Option B) or using REGEX (Option C), as it leverages the platform's native globalization engine. Option D (FOR VIEW) is used to update the "Last Viewed" date and does not affect the visual formatting of the returned data. By using FORMAT(), the developer ensures that the Lightning component remains lightweight and consistent with the standard Salesforce user experience.

NEW QUESTION: 52
Java
@isTest
static void testUpdateSuccess() {
Account acet = new Account(Name = 'test');
insert acet;
// Add code here
extension.inputValue = 'test';
PageReference pageRef = extension.update();
System.assertNotEquals(null, pageRef);
}
What should be added to the setup, in the location indicated, for the unit test above to create the controller extension for the test?
A. AccountControllerExt extension = new AccountControllerExt(acet);
B. ApexPages.StandardController sc = new ApexPages.StandardController(acet);
AccountControllerExt extension = new AccountControllerExt(sc);
C. ApexPages.StandardController sc = new ApexPages.StandardController(acet.Id);
AccountControllerExt extension = new AccountControllerExt(sc);
D. AccountControllerExt extension = new AccountControllerExt(acet.Id);
Answer: B (LEAVE A REPLY)
123

To properly test a Visualforce Controller Extension in Apex, the developer must replicate the instantiation process used by the Lightning Platform at runtime. A4 controller extension is a class whose constructor is specifically defined to accept a single parameter of the type ApexPages.StandardController. 5The standard controller acts as the bridge b6etween the custom extension logic and the underlying Salesforce record (in this case, an Account).
In the provided test method, a record acet has been inserted into the database. To create the extension object, the developer must first instantiate the StandardController by passing the acet SObject record into its constructor. Once the standard controller instance (e.g., sc) is created, it is then passed into the constructor of the AccountControllerExt class. Option B is the correct implementation of this two-step pattern. Option C is incorrect because the StandardController constructor requires a full SObject record, not just an ID string.
Options A and D are incorrect because extension classes do not support constructors that take an SObject or an ID directly; they specifically require the StandardController object to provide the necessary context for standard actions and fields. Following the correct pattern ensures the test accurately simulates the page execution context.

NEW QUESTION: 53
As part of a custom interface, a developer team creates various new Lightning web components. Each of the components handles errors using toast messages. During acceptance testing, users complain about the long chain of toast messages that display when errors occur loading the components. Which two techniques should the developer implement to improve the user experience?
A. Use a Lightning web component to aggregate and display all errors.
B. Use the window.alert() method to display the error messages.23
C. Use a <template> tag to display in-place error messages.24
D. Use public properties on each component to display the error messages.25
Answer: A,C (LEAVE A REPLY)
When multiple components on a single page all fail simultaneously (for example, due to a shared service failure), individual toast messages stack up, creating a poor user experience. To resolve this, developers should shift away from "ephemeral" notifications like toasts toward more structured error handling.
Option A involves creating a dedicated error-handling component. This component can act as a listener or a central repository for errors across the page. Instead of each component firing its own toast, they can communicate their error state to this central component, which aggregates the messages into a single, clean display. This reduces visual clutter and allows the user to read all errors in one place.
Option C refers to "in-place" error handling. Using conditional rendering (the lwc:if or template if:true directives), a component can hide its normal UI and display an error message exactly where the component would have been. This provides immediate context to the user about which specific part of the page failed without interrupting the overall flow with pop-ups.

Option B (window.alert) is considered a legacy practice that blocks the browser thread and is generally avoided in modern web development. Option D (public properties) is a mechanism for component communication but doesn't solve the display problem itself. Combining A and C provides a modern, professional, and less intrusive error-handling strategy.

NEW QUESTION: 54
Consider the following code snippet:
Java
HttpRequest req = new HttpRequest();
req.setEndpoint('https://TestEndpoint.example.com/some_path');
req.setMethod('GET');
Blob headerValue = Blob.valueOf('myUserName' + ':' + 'strongPassword'); String authorizationHeader = 'BASIC ' + EncodingUtil.base64Encode(headerValue);
req.setHeader('Authorization', authorizationHeader); Http http = new Http();
HTTPResponse res = http.send(req); Which two steps should the developer take to add flexibility to change the endpoint and credentials without needing to modify code?1
A. Use req.setEndpoint(Label.endPointURL);
B. Use req.setEndpoint('callout:endPoint_NC'); within the callout request.2
C. Store the URL of the3 endpoint in a custom Label named endPointURL.4
D. Create a Named Credential, endPoint_NC, to store the endpoint and credentials.5
Answer: B,D (LEAVE A REPLY)
7
The provided code uses hard-coded strings for the URL and authe8ntication headers, which is a security risk and makes maintenance difficult. To add flexibility and security, Salesforce provides Named Credentials (Option D). A Named Credential acts as a single definition that encapsulates both the endpoint URL and the required authentication (Basic, OAuth, etc.) in a single Setup record.
By using the callout: protocol in the setEndpoint method (Option B), the developer instructs the Apex runtime to look up the configuration stored in the Named Credential named endPoint_NC. This approach provides several key benefits:
* Security: Credentials like passwords are encrypted and stored safely in the platform's metadata, not in the code.
* Simplified Code: The logic for building the Authorization header and base64 encoding (as seen in the original snippet) is handled automatically by the platform.
* Maintainability: If the endpoint URL or the password changes, an administrator can update the Named Credential record via the UI without requiring any code deployment.
Custom Labels (Option A and C) can store URLs, but they cannot securely handle authentication logic or headers. Therefore, the combination of a Named Credential and the callout: syntax is the optimal platform- native solution for flexible and secure integrations.

NEW QUESTION: 55
An Apex trigger and Apex class increment a counter, `Edit_Count__c`, any time the Case is changed.
```java
public class CaseTriggerHandler {
public static void handle(List<Case> cases) {
for (Case c : cases) {
c.Edit_Count__c = c.Edit_Count__c + 1;
}
}
}
trigger on Case(before update) {
CaseTriggerHandler.handle(Trigger.new);
}
```
A new before-save record-triggered flow on the Case object was just created in production for when a Case is created or updated. Since the process was added, there are reports that `Edit_Count__c` is being incremented more than once for Case edits. Which Apex code fixes this problem?
A. ```java
public class CaseTriggerHandler {
public static Boolean firstRun = true;
public static void handle(List<Case> cases) {
for (Case c : cases) {
B. Edit_Count__c = c.Edit_Count__c + 1;
}
}
}
trigger on Case(before update) {
CaseTriggerHandler.firstRun = true;
if (CaseTriggerHandler.firstRun) {
CaseTriggerHandler.handle(Trigger.newMap);
}
CaseTriggerHandler.firstRun = false;
}
```
**C.** ```java
public class CaseTriggerHandler {
public static Boolean firstRun = true;
public static void handle(List<Case> cases) {
for (Case c : cases) {
```

D. Edit_Count__c = c.Edit_Count__c + 1;
}
}
}
trigger on Case(before update) {
if (CaseTriggerHandler.firstRun) {
CaseTriggerHandler.handle(Trigger.new);
}
CaseTriggerHandler.firstRun = false;
}
```
**E.** ```java
public class CaseTriggerHandler {
Boolean firstRun = true;
public static void handle(List<Case> cases) {
if (firstRun) {
for (Case c : cases) {
**F.** Edit_Count__c = c.Edit_Count__c + 1;
}
}
firstRun = false;
}
}
trigger on Case(before update) {
CaseTriggerHandler.handle(Trigger.new);
}
```
G. ```java
trigger on Case(before update) {
Boolean firstRun = true;
if (firstRun) {
CaseTriggerHandler.handle(Trigger.newMap);
}
firstRun = false;
}
```

**Answer: B (LEAVE A REPLY)**

This scenario illustrates a classic trigger recursion issue triggered by the Salesforce Order of Execution. When a record is updated, Salesforce runs through a specific sequence: first, it executes "Before-Save" Record- Triggered Flows, then "Before" triggers, followed by "After" triggers, and finally "After-Save" flows and processes. If a process or flow updates
```

the same record that initiated the transaction, the entire cycle of Apex triggers can be re-invoked within that single transaction.
To prevent logic from running multiple times during these re-entrant cycles, developers implement a static boolean variable as a "recursion guard." Static variables in Apex persist for the entire duration of a single transaction. By checking the value of a static boolean at the start of the trigger, the code can determine if it has already processed the current set of records.
Option B is the correct implementation of this pattern. It defines `public static Boolean firstRun = true;` in the handler class. The trigger checks if `firstRun` is true, executes the handler logic, and then immediately sets
`firstRun` to false. Any subsequent execution of the Case trigger within the same transaction will find the variable set to false and skip the increment logic.
Option A is incorrect because it resets `firstRun` to true every time the trigger starts, defeating the guard.
Option C incorrectly uses an instance variable (non-static), which is recreated for every call. Option D uses a local variable within the trigger body, which is re-initialized to true every time the trigger fires, providing no protection against recursion.

NEW QUESTION: 56
Refer to the component code and requirements below:
HTML
<lightning:layout multipleRows="true">
<lightning:layoutItem size="12">{!v.account.Name}</lightning:layoutItem>
<lightning:layoutItem size="12">{!v.account.AccountNumber}</lightning:layoutItem>
<lightning:layoutItem size="12">{!v.account.Industry}</lightning:layoutItem>
</lightning:layout>
Requirements:
* For mobile devices, the information should display in three rows.
* For desktops and tablets, the information should display in a single row.
Requirement 2 is not displaying as desired. Which option has the correct component code to meet the requirements for desktops an7d tablets?
A. HTML
<lightning:layout multipleRows="true">
<lightning:layoutItem size="12" mediumDeviceSize="6">{!
v.account.Name}</lightning:layoutItem>
<lightning:layoutItem size="12" mediumDeviceSize="6">{!
v.account.AccountNumber}</lightning:
layoutItem>
<lightning:layoutItem size="12" mediumDeviceSize="6">{!
v.account.Industry}</lightning:layoutItem>
</lightning:layout>

B. <lightning:layout multipleRows="true"></lightning:layout>1213
C. 1415
HTML
<lightning:layout multipleRows="true">
<lightning:layoutItem size="12" largeDeviceSize="4">{!
v.account.Name}</lightning:layoutItem>
<lightning:layoutItem size="12" largeDeviceSize="4">{!
v.account.AccountNumber}</lightning:
layoutItem>
<lightning:layoutItem size="12" largeDeviceSize="4">{!
v.account.Industry}</lightning:layoutItem>
</lightning:layout>
D. HTML
<lightning:layout multipleRows="true">
<lightning:layoutItem size="12" mediumDeviceSize="4" largeDeviceSize="4"> {!
v.account.Name} <
/lightning:layoutItem>
<lightning:layoutItem size="12" mediumDeviceSize="4" largeDeviceSize="4"> {!v.account.
AccountNumber} </lightning:layoutItem>
<lightning:layoutItem size="12" mediumDeviceSize="4" largeDeviceSize="4"> {!
v.account.Industry}
</lightning:layoutItem>
</lightning:layout>
Answer: (SHOW ANSWER)
To achieve a responsive layout in Salesforce Aura components, the lightning:layout and lightning:layoutItem components utilize a 12-column grid system. The size attribute defines the default behavior, which targets the smallest devices (mobile). In the original code, size="12" is applied to all three items. Since each item occupies the full 12 columns, they stack vertically in three rows.
To meet Requirement 2 (displaying in a single row for tablets and desktops), the three items must share the
12-column horizontal space. Mathematically, $12 \div 3 = 4$. Therefore, each item must be assigned a size of
4 for larger screen breakpoints.
Option D is the correct implementation. It maintains size="12" for mobile (Requirement 1) and sets mediumDeviceSize (tablets) and largeDeviceSize (desktops) to 4. This ensures that on any screen larger than a phone, the three items will sit side-by-side in a single row.
Option A is incorrect because mediumDeviceSize="6" would only allow two items per row ($6 + 6 = 12$), forcing the third item to a second row. Option C is incomplete as it only specifies the largeDeviceSize, potentially leaving tablets in a stacked configuration. Option

D follows the best practice of explicitly defining the span for all relevant device categories to ensure a consistent user experience.

NEW QUESTION: 57
A developer has a test class that creates test data before making a mock callout but now receives a "You have uncommitted work pending. Please commit or rollback before calling out" error. Which step should be taken to resolve the error?
A. Ensure both the insertion and mock callout occur after the Test.stopTest().1
B. Ensure the records are inserted before the Test.startTest() statement and the mock callout occurs within a method annotated with @testSetup.2
C. Ensure the records are inserted befo3re the Test.startTest() statement and the mock callout occurs after the Test.startTest().45
D. Ensure both the insertion and mock callout occur after the Test.startTest().67
Answer: C (LEAVE A REPLY)
Comprehensive and Deta10iled 150 to 250 words of E11xplanation:
This error occurs because Salesforce prohibits making a callout (even a mock one) in the same transaction after a DML operation has been performed. When you insert test data, it creates a "pending" transaction in the database. If you then attempt to execute a callout immediately, the platform throws the CalloutException to prevent data inconsistency issues.
To resolve this in a test context, you must separate the DML operation from the callout using the Test.
startTest() and Test.stopTest() methods. When Test.startTest() is called, Salesforce provides a fresh set of governor limits and effectively creates a new transaction context for the code that follows. By inserting the records before Test.startTest() and performing the logic that triggers the callout after Test.startTest(), the developer ensures that the DML operation is "committed" to the test database context before the callout is initiated.
Option C correctly identifies this pattern. Option B is incorrect because @testSetup is used for global data creation across all test methods and does not specifically address the callout transaction split. Options A and D do not solve the problem because they either keep the DML and callout in the same context or place the DML in a context where it would still interfere with the subsequent callout request.

NEW QUESTION: 58
The Account after-update trigger fires whenever an Account's address is updated, and it updates every associated Contact with that address. The Contact after-update trigger fires on every edit, and it updates every Campaign Member record related to the Contact with the Contact's state. Consider the following: A mass update of 200 Account records' addresses, where each Account has 50 Contacts. Each Contact has one Campaign Member. This means there are 10,000 Contact records across th23e Accounts and 10,000

Campaign Member records across the contacts. What will happen when the mass update occurs?24252627

A. The mass update of Account address 28will succeed, but the Contact address updates wil29l fail due to exceeding number of records processed by DML statements.3031

B. There will be no32 error, since each trigger fires 33within its own context and each trigger does not exceed the limit of the number of records processed by DML statements.34

C. The mass update will fail, since the two triggers fire in the same35 context, thus exceeding the number of records.36

Answer: (SHOW ANSWER)

38

In Salesforce, all triggers that fire as a result of an initial DML operation ru39n within the same execution context and the same transaction. This means that all governor limits-such as the limit of 10,000 records processed by DML statements-are shared across the entire chain of events.

In this scenario, the transaction begins with the update of 200 Accounts. The Account trigger then attempts to update 10,000 related Contacts (50 Contacts per Account across 200 Accounts). This update of 10,000 Contacts then fires the Contact trigger, which attempts to update 10,000 related Campaign Members. When the system attempts to process these updates, the total number of records processed via DML in a single transaction will exceed the platform limit of 10,000 rows.

Even though the Account update is only 200 records, the subsequent cascading updates (10,000 Contacts +

10,000 Campaign Members) bring the total DML count to 20,200. This triggers a LimitException, and because Apex transactions are atomic, the entire operation will fail and roll back. To resolve this, the developer would need to handle these updates asynchronously (e.g., using Batch Apex or Queueable) to split the cascading updates into separate transactions with their own sets of governor limits.

NEW QUESTION: 59

Consider the following code snippet:

HTML

```
<c-selected-order>
<template for:each={orders.data} for:item="order">
<c-order orderId={order.Id}></c-order>
</template>
</c-selected-order>
```

How should the <c-order> component communicate to the <c-selected-order> component that an order has been selected by the user?

A. Create and dispatch a custom event.

B. Create and fire a component event.

C. Create and fire an application event.
D. Create and fire a standard DOM event.
Answer: A (LEAVE A REPLY)
In Lightning Web Components (LWC), components communicate up the containment hierarchy (from child to parent) by creating and dispatching Custom Events.
Unlike the older Aura framework, which used specific "Component" or "Application" event types (Options B and C), LWC relies on standard web browser event protocols. The child component (c-order) would instantiate a CustomEvent object, optionally including a data payload in the detail property (such as the orderId), and then call the this.dispatchEvent() method.
The parent component (c-selected-order) then listens for this event using an on prefix (e.g., onselection=
{handleSelection}) in its HTML template. While you could technically use a standard DOM event (Option D), the CustomEvent interface is the platform-standard best practice in LWC for passing custom data payloads between components. Custom events allow for clean encapsulation and follow the modern "Events Up, Properties Down" design pattern common in reactive frameworks like LWC, React, and Vue.

NEW QUESTION: 60
Which three Visualforce components can be used to initiate Ajax behavior to perform partial page updates?
A. <apex:form>
B. <apex:actionStatus>
C. <apex:commandButton>
D. <apex:actionSupport>
E. <apex:commandLink>
Answer: C,D,E (LEAVE A REPLY)
Comprehensive and Detailed Explanation:
In Visualforce, partial page updates (AJAX) are achieved using the reRender attribute. This attribute allows a component to refresh only a specific part of the page identified by an ID, rather than performing a full browser reload.
* <apex:commandButton> (Option C) and <apex:commandLink> (Option E) are standard action components. When their reRender attribute is populated, they perform an asynchronous postback.
* <apex:actionSupport> (Option D) is an auxiliary component that adds AJAX functionality to other components that do not natively support it. For example, you can place an actionSupport inside an inputText to trigger a partial refresh on the onchange event.
Option A (<apex:form>) is a container and does not initiate AJAX behavior itself. Option B (<apex:
actionStatus>) is used to display the status of an AJAX request (e.g., a loading spinner) but does not initiate the request.

NEW QUESTION: 61
A developer needs to store variables to control the style and behavior of a Lightning Web Component. Which feature can be used to ensure that the variables are testable in both Production and all Sandboxes?
A. Custom metadata
B. Custom object
C. Custom setting
D. Custom variable
Answer: (SHOW ANSWER)
For application configurations that control behavior and styling across different environments, Custom Metadata Types (Option A) are the industry-standard choice. The primary advantage of Custom Metadata over Custom Settings or Custom Objects is tha1t the2 records themselves are treated as metadata. This means they can be deployed using Change Sets, Salesforce CLI, or packages. When a developer moves code from a Sandbox to Production, the associated configuration records are included in the deployment, ensuring that the Lightning Web Component behaves identically in both environments without manual data setup.
Furthermore, Custom Metadata is highly "testable." In Apex unit tests, you can query Custom Metadata records without needing the (SeeAllData=true) annotation. If your component's Apex controller logic depends on these variables, the tests will remain robust and environment-independent. Custom Settings (Option C) and Custom Objects (Option B) store records as data, which cannot be deployed via Change Sets and require manual entry or data loading in every new sandbox or production org, increasing the risk of configuration errors. Custom Metadata provides the most "extendable" and deployable way to manage component-level constants and behavioral flags.

NEW QUESTION: 62
A developer has business logic in a trigger that flags high-value opportunities. There is a new requirement to also display high value opportunities in a Lightning web component. Which two steps should the developer take to meet these business requirements, and also

prevent the business logic that identifies high-value opportunities from being repeated in more than one place?

A. Call the trigger from the Lightning web component.

B. Leave the business logic code inside the trigger for efficiency.

C. Create a helper class that fetches the high value opportunities.

D. Use custom metadata to hold the high value amount.

Answer: (SHOW ANSWER)

To adhere to the DRY (Don't Repeat Yourself) principle and ensure maintainability, business logic should be decoupled from specific execution contexts like triggers.

Option C is a best practice. By moving the logic into a Service or Helper class, the code becomes reusable.

The trigger can call this class during DML operations to flag records, and an Apex controller (invoked by the Lightning Web Component) can call the same class to retrieve the high-value records for display. This ensures that if the criteria for "high-value" changes, the developer only needs to update the code in one location.

Option D complements this by externalizing the "High Value Amount" threshold. Instead of hardcoding a value (e.g., $100,000) within the Apex code, storing it in Custom Metadata allows administrators to adjust the threshold without requiring a code deployment. The helper class can dynamically query this metadata.

Option A is technically impossible; triggers are fired by the database system in response to DML, not called directly by UI components. Option B leads to "Trigger Bloat" and prevents the LWC from accessing the logic, forcing duplication. Together, C and D provide a scalable, configurable, and reusable architecture.

NEW QUESTION: 63

A company needs to automatically delete sensitive information after seven years. This could delete almost a million records every day. How can this be achieved?

A. Schedule an @future process to query records older than seven years, and then recursively invoke itself in 1,000 record batches to delete them.

B. Use aggregate functions to query for records older than seven years, and then delete the AggregateResult objects.

C. Perform a SOSL statement to find records older than 7 years, and then delete the entire result set.

D. Schedule a batch Apex process to run every day that queries and deletes records older than seven years.

Answer: D (LEAVE A REPLY)

When dealing with large-scale data maintenance, such as deleting a million records daily, Batch Apex is the most robust and scalable solution provided by the Salesforce platform.

Batch Apex is specifically designed for processing high volumes of records by breaking the total record set into manageable "batches" (defaulting to

200 records per batch). This prevents the transaction from hitting platform governor limits, such as the limit on the total number of DML statements or the maximum number of records processed in a single transaction.
By scheduling a Batch class, the system handles the heavy lifting asynchronously, ensuring that org performance is not negatively impacted during peak hours.
Other options are unsuitable for this scale. Using @future methods for recursive processing is a poor practice because Salesforce prohibits calling a future method from another future method, and it is difficult to monitor or manage the execution flow.
Aggregate functions and AggregateResult o1bjects are used for calculations and data grouping, not for DML operations like deletion. Similarly, SOSL is optimized for text-based searching across multiple objects and returns a limited number of results, making it inappropriate for an exhaustive cleanup of a million records. Batch Apex allows for the use of a QueryLocator, which can handle up to 50 million records, providing the necessary throughput for this high-volume requirement.

NEW QUESTION: 64
An org records customer order information in a custom object, Order__c, that has fields for the shipping address. A developer is tasked with adding code to calculate shipping charges on an order, based on a flat percentage rate associated with the region of the shipping address. What should the developer use to store the rates by region, so that when the changes are deployed to production no additional steps are needed for the calculation to work?3132
A. Custom metadata type3334
B. Custom list setting3536
C. Custom object3738
D. Custom hierarchy setting3940
Answer: (SHOW ANSWER)
Comprehensive and Detailed 43150 to 250 words of Explanation:44
The critical requirement in this scenario is that "no additional steps are needed45" upon deployment. This refers to the ability to migrate not just the logic, but also the configuration data (the rates) across environments (e.g., from Sandbox to Production).
Custom Metadata Types (Option A) are the only option among those listed that are treated as metadata rather than data. This means that individual records within a Custom Metadata Type can be included in Change Sets, unlocked packages, or via the Metadata API. When the developer deploys the code, they include the metadata records for the regions and rates in the same package. Once the deployment finishes, the code can immediately query those rates and perform calculations without any manual record creation in the production environment.
In contrast, Custom List Settings (Option B), Custom Objects (Option C), and Hierarchy Settings (Option D) store their records as "data." Data is not deployable via Change Sets. If a developer used these options, they would have to manually re-enter all the regional

rates in Production after the code was deployed, which violates the requirement for a seamless, automated deployment process.

NEW QUESTION: 65
An org has a requirement that an Account must always have one and only one Contact listed as Primary. So selecting one Contact will de-select any others. The client wants a checkbox on the Contact called 'Is Primary' to control this feature. The client also wants to ensure that the last name of every Contact is stored entirely in uppercase characters. What is the optimal way to implement these requirements?12345
A. Write a Validation Rule on the Contact for the Is Primary logic and a before upd6ate trigger on Contact for the last name lo7gic.8910
B. Write an after update trigger on Conta11ct for the Is Primary logic and a separate befo12re update trigger on Contact for th13e last name logic.
C. Write an after update trigger on Account for the Is Primary logic and a before update trigger on Contact for the last name logic.
D. Write a single trigger on Contact for both after update and before update and callout to helper classes to handle each set of logic.14
Answer: (SHOW ANSWER)
16
The optimal architectural ap17proach for complex logic on a single object is to use a single trigger per object that delegates responsibilities to a helper class. This requirement involves two distinct types of logic: field manipulation (uppercase names) and cross-record updates (ensuring only one primary contact).
In Salesforce, Before triggers are ideal for updating fields on the record that initiated the trigger, as the changes are saved to the database without an additional DML statement. Therefore, converting the LastName to uppercase should occur in a before update and before insert context. Conversely, the "Is Primary" logic requires updating other records (de-selecting other contacts on the same account). This must happen in an After trigger context to ensure the primary record has been successfully updated and to avoid recursion or conflicts with the initial save.
Option D is the best practice because it follows the "One Trigger Per Object" design pattern. By using a single trigger with multiple context variables (isBefore, isAfter, isUpdate), the developer can cleanly route the "Last Name" logic to a before-save helper and the "Is Primary" cross-record update logic to an after-save helper.
This centralized control prevents multiple triggers from firing in an unpredictable order and simplifies maintenance and debugging.

NEW QUESTION: 66
A developer is developing a reusable Aura component that will reside on an sObject Lightning page with the following HTML snippet:
HTML

<aura:component implements="force:hasRecordId,flexipage:availableForAllPageTypes">
Hello!
</aura:component>
How can the component's controller get the context of the Lightning page that the sObject is on without requiring additional test coverage?
A. Create a design attribute and configure via App Builder.
B. Set the sObject type as a component attribute.
C. Use the getSObjectType() method in an Apex class.
D. Add force:hasSObjectName to the implements attribute.
Answer: (SHOW ANSWER)
In Aura components, Salesforce provides specific interfaces that "inject" context into the component automatically. The developer is already using force:hasRecordId, which automatically populates a recordId attribute with the ID of the current record.
To get the API Name of the sObject (e.g., "Account" or "Contact") without writing Apex code or requiring manual configuration in the App Builder, the developer should implement force:hasSObjectName (Option D). When this interface is added to the implements attribute, the framework automatically provides an attribute named sObjectName to the component.
This is the most efficient solution because:
* It is purely declarative and handled by the Lightning framework.
* It requires zero Apex code, meaning there is no need for additional unit tests or code coverage (satisfying the requirement "without requiring additional test coverage").
* It makes the component truly reusable; you can drop it on an Account page, a Contact page, or any custom object page, and it will immediately know which object it is currently displaying.
Options A and B require manual configuration or hardcoding, which reduces reusability. Option C requires an Apex call, which adds complexity and necessitates writing test classes for coverage.

NEW QUESTION: 67
The Account object has a field, Audit_Code__c, that is used to specify what type of auditing the Account needs and a Lookup to User, Auditor__c, that is the assigned auditor. When an Account is initially created, the user specifies the Audit_Code__c. Each User in the org has a unique text field, Audit_Code__c, that is used to automatically assign the correct user to the Account's Auditor__c field.
Trigger:
Java
trigger AccountTrigger on Account (before insert) {
AuditAssigner.setAuditor(Trigger.new);
}
Apex Class:

```
Java
public class AuditAssigner {
public static void setAuditor(List<Account> accounts) {
for (User u : [SELECT Id, Audit_Code__c FROM User]) {
for (Account a : accounts) {
if (u.Audit_Code__c == a.Audit_Code__c) {
a.Auditor__c = u.Id;
}
}
}
}
}
```

What should be changed to most optimize the code's efficiency?

A. Add a WHERE clause to the SOQL query to filter on audit codes.

B. Build a Map<String, List<Account>> of audit code to accounts.

C. Build a Map<Id, List<String>> of Account Id to audit codes.

D. Add an initial SOQL query to get all distinct audit codes.

Answer: A (LEAVE A REPLY)

The current implementation suffers from a significant performance bottleneck due to an unfiltered SOQL query and a nested loop ($O(N \times M)$ complexity). The query [SELECT Id, Audit_Code__c FROM User] retrieves every single user in the Salesforce organization. In large enterprises with thousands of users, this results in excessive heap memory usage and consumes unnecessary CPU time, even if only a few specific users are needed for the current batch of accounts.

The most critical optimization to improve efficiency is to add a WHERE clause (Option A) to the SOQL query. By first collecting the Audit_Code__c values from the accounts list into a Set<String> and then using that set in the query (e.g., WHERE Audit_Code__c IN :auditCodeSet), the developer ensures that Salesforce only retrieves the relevant User records from the database. This significantly reduces the data volume transferred from the database to the application server and avoids hitting governor limits like the "Total number of SOQL records retrieved." While building a Map (Option B) is also a best practice to eliminate the nested loop and move toward $O (N+M)$ complexity, the primary efficiency gain in terms of platform resources and governor limit safety comes from making the query selective. A selective query is the foundation of a "bulkified" and performant trigger handler.

NEW QUESTION: 68

To avoid duplicating code and improve maintainability, how should Universal Containers implement an API integration for code reuse?

A. Create a reusable Apex class for the API integration and invoke it from the relevant Apex classes.

B. Use a separate Apex class for each API endpoint to encapsulate the integration logic.
C. Store the API integration code as a static resource and reference it in each Apex class.
D. Include the API integration code directly in each Apex class that requires it.
Answer: A (LEAVE A REPLY)
Comprehensive and Detailed Explanation:1
The fundamental principle of DRY (Don't Repeat Yourself) in Apex development dictates that logic used in multiple places should be centralized. For API integrations, this typically invo2lves creating a Utility or Service Class (Option A).
This class handles the common "plumbing" of the integration:
* Retrieving Named Credentials.
* Setting headers (Content-Type, Timeout).
* Standardizing error handling and logging.
* Serializing and deserializing JSON payloads.
By invoking this central class from various triggers, batch jobs, or controllers, the developer ensures that any change to the API (such as a version update or header change) only needs to be made in one place. Option B leads to "class sprawl" and duplicate boilerplate code. Option C is incorrect as Static Resources cannot contain executable Apex code. Option D is the definition of poor maintainability.

NEW QUESTION: 69
A company has an Apex process that makes multiple extensive database operations and web service callouts.
The database processes and web services can take a long time to run and must be run sequentially. How should the developer write this Apex code without running into governor limits and system limitations?123
A. Use Apex Scheduler to schedul4e each process.56
B. Use Limits class to stop entire pr7ocess once governor limits are reached.8
C. Use multip9le @future methods for each process and callout.
D. Use Queueable Apex to chain the jobs to run sequentially.
Answer: (SHOW ANSWER)
This requirement specifies two critical constraints: the operations take a long time (likely exceeding synchronous CPU and timeout limits) and they must be performed sequentially. Queueable Apex (Option D) is the optimal solution because it supports job chaining. Chaining allows one Queueable job to enqueue another Queueable job from its execute() method. This effectively creates a sequence where Job B only starts after Job A has successfully finished. Each job in the chain starts with a fresh set of governor limits, which is essential for "extensive database operations" and long-running callouts.
Option C (@future) is unsuitable because future methods cannot be chained; you cannot call one future method from another. Furthermore, the order in which multiple future methods execute is not guaranteed by the platform, violating the "sequential" requirement.
Option A (Apex Scheduler) is intended for delayed or recurring tasks and is not designed

for managing immediate sequential dependencies. Option B is a defensive coding practice but does not solve the underlying need to complete the work. Queueable Apex provides the programmatic control needed to manage complex, multi-step asynchronous workflows safely.

NEW QUESTION: 70
A developer writes a Lightning web component that displays a dropdown list of all custom objects in the org.
An Apex method prepares and returns data to the component. What should the developer do to determine which objects to include in the response?
A. Import the list of all custom objects from @salesforce/schema.
B. Check the getObjectType() value for 'Custom' or 'Standard' on the sObject describe result.
C. Check the isCustom() value on the sObject describe result.
D. Use the getCustomObjects() method from the Schema class.
Answer: C (LEAVE A REPLY)
Comprehensive and Detailed Explanation:
To programmatically identify custom objects in Apex, developers utilize Schema Describe information. The Schema.getGlobalDescribe() method returns a map of all object tokens in the org. By iterating through these tokens and calling the getDescribe() method on each, the developer can access various properties of the object's metadata.
The specific property to distinguish between standard and custom objects is the isCustom() method (Option C). This boolean method returns true if the object is a custom object (including custom settings and custom metadata types) and false if it is a standard platform object.
Option A is incorrect because @salesforce/schema is used in LWC to import references to specific known objects/fields, not to discover the entire org's metadata dynamically. Option B is incorrect as getObjectType() returns an SObjectType token, not a string value like 'Custom'. Option D is incorrect because there is no getCustomObjects() method in the standard Schema class; discovery must be done via the global describe map.

NEW QUESTION: 71
A company has reference data stored in multiple custom metadata records that represent default information and delete behavior for certain geographic regions. When a contact is inserted, the default information should be set on the contact. Additionally, if a user attempts to delete a contact that belongs to a flagged region, the user must get an error message. Depending on company personnel resources, what are two ways to automate this?16
A. Remote action17
B. Flow Builder18
C. Apex invocable method19

D. Apex trigger20

Answer: B,D (LEAVE A REPLY)

Comprehensi22ve and Detailed 150 to 250 words of Explanation:

This requirement involves two different automation triggers: record creation (insert) and record deletion (delete). To handle these events, Salesforce provides both declarative and programmatic options.

Flow Builder (Option B) is the preferred declarative tool. Record-Triggered Flows can be configured to run

"Before" a record is saved to handle the default value assignment upon insertion. They can also be configured to run when a record is deleted. Using the "Custom Error" element within a Flow, an administrator can easily block a deletion and present a user-friendly error message if the contact belongs to a flagged region.

Apex Triggers (Option D) are the programmatic equivalent. A developer can write a before insert trigger to perform the metadata lookup and set field defaults, and a before delete trigger to check the region and call addError() on the record to prevent the deletion.

Options A (Remote Action) and C (Invocable Method) are not standalone automation triggers. A Remote Action is for JavaScript-to-Apex communication in Visualforce, and an Invocable Method is a piece of code called by another tool like a Flow or Strategy Builder.

Therefore, Flow and Triggers are the two primary mechanisms to satisfy both the insertion and deletion requirements.

NEW QUESTION: 72

Part of a custom Lightning component displays the total number of Opportunities in the org, which are in the millions. The Lightning component uses an Apex method to get the data it needs. What is the optimal way for a developer to get the total number of Opportunities for the Lightning component?

A. COUNT () SOQL aggregate query on the Opportunity object

B. Apex batch job that counts the number of Opportunity records

C. SUM () SOQL aggregate query on the Opportunity object

D. SOQL for loop that counts the number of Opportunities records

Answer: A (LEAVE A REPLY)

When you need to retrieve the total count of records in a large dataset (millions of records), a SOQL aggregate query using COUNT() (Option A) is the most efficient and performant method. Salesforce optimizes aggregate functions at the database 13level. Unlike a standard query that returns in14dividual records and counts against the "50,000 SOQL rows" limit, a COUNT() query returns a single integ15er result and counts as only one row to16ward the governor limits.17 This allows a developer to count millions of records in a single synchronous transactio18n without hitting row limits or causing significant CPU time issues. Option D (SOQL for loop) is the worst approach, as it would attempt to load every individual record into memory, hitting the 50,000-row limit almost immediately and likely causing a LimitException or Request Timeout.

Option B (Batch Apex) is unnecessary for a simple count and is much slower because it runs asynchronously.
Option C (SUM) is used for adding up values in numeric fields, not for counting the number of records. For high-volume record counting, SELECT COUNT(Id) FROM Opportunity is the platform-standard approach to provide data to a Lightning component efficiently.

NEW QUESTION: 73
What are three reasons that a developer should write Jest tests for Lightning web components?
A. To test basic user interaction
B. To test a component's non-public properties
C. To test how multiple components work together
D. To verify the DOM output of a component
E. To verify that events fire when expected
Answer: A,D,E (LEAVE A REPLY)
Jest is a powerful JavaScript testing framework used for Lightning Web Components (LWC) to ensure individual units of code function correctly in isolation. One of the primary reasons to use Jest is to verify the DOM output of a component (D). Since LWC is a UI framework, Jest allows developers to inspect the rendered HTML to ensure that elements, classes, and data are displayed as intended after a state change.
Furthermore, Jest is essential for testing basic user interaction (A). Developers can simulate events like button clicks or text input and then assert that the component's state or UI updates accordingly.
Another critical use case is to verify that events fire when expected (E). Components often communicate with parents via custom events; Jest allows you to "spy" on these events to ensure they are dispatched with the correct detail payload at the right time. Conversely, Jest is not intended for testing how multiple components work together (C)-this is the domain of integration tests or end-to-end tests (like UTAM). Additionally, Jest tests should focus on the component's public API and observable behavior; testing non-public (private) properties (B) is generally discouraged as it leads to brittle tests that break upon internal refactoring. By focusing on DOM output, interactions, and events, Jest provides a fast, reliable way to maintain component quality.

NEW QUESTION: 74
Given the following code:
Java
for (Contact c : [SELECT Id, LastName FROM Contact WHERE CreatedDate = TODAY])
{
Account a = [SELECT Id, Name FROM Account WHERE CreatedDate = TODAY LIMIT 5];
c.AccountId = a.Id; update c;
}

Assuming there were 10 Contacts and five Accounts created today, what is the expected result?

A. System.QueryException: List has more than one row for Assignment on Account

B. System.LimitException: Too many SOQL Queries on Contact

C. System.LimitException: Too many SOQL Queries on Account

D. System.QueryException: Too many DML Statement errors on Contact

Answer: A (LEAVE A REPLY)

2

The primary issue in this code snippet is a violation of basic variable assignment rules in Apex when dealing with SOQL results. In the line Account a = [SELECT Id, Name FROM Account ...], the code attempts to assign the result of a query directly 3to a single SObject variable (Account a).

In Apex, a SOQL query assigned to a single SObject variable must return exactly one record. If the query returns zero records, it throws a System.QueryException: List has no rows for assignment. If the query returns more than one record, it throws a System.QueryException: List has more than one row for assignment (Option A). Since the prompt explicitly states that five Accounts were created today and the query is not filtered to a single unique ID, the query will return five records. Even though there is a LIMIT 5 clause, that only caps the results at five; it does not ensure only one is returned. To fix this, the result should be assigned to a List<Account> or the query should be filtered to return exactly one row.

While the code also violates the "SOQL in a loop" best practice, it would not hit the LimitException (Option C) in this specific case because the loop only runs 10 times (10 contacts), and the limit is 100 queries. The runtime assignment error occurs before any governor limits are breached.