Free Salesforce Plat-Dev-301 Practice Questions

Shared by Nielsen on 13-08-2026

For More Free Questions and Preparation Resources

Check the Links on Last Page

Question 1

Question Type: MultipleChoice

The use of the transient keyword in Visualforce page helps with which performance issue?

Options:

A- Reduces view state
B- Improves query performance
C- Reduces load times
D- Improves page transfers

Answer:

A

Explanation:

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.

Question 2

Question Type: MultipleChoice

A company wants to track revenue through a related object. They need to perform a one-time seeding of data for roughly 100,000 Opportunities, creating Revenue records based on complex logic. What is the optimal way to automate this?

Options:

A- Use Database.executeBatch() to invoke a Database.Batchable class.
B- Use System.enqueueJob() to invoke a Queueable class.
C- Use Database.executeBatch() to invoke a Queueable class.
D- Use System.scheduleJob() to schedule a Database.Scheduleable class.

Answer:

A

Explanation:

For high-volume data processing (such as seeding 100,000 records) involving complex logic, Batch Apex (Option A) is the standard and most robust solution. Batch Apex is designed to handle up to 50 million records by breaking the total set into smaller, manageable chunks (defaulting to 200 records per batch).

Each batch execution gets its own set of governor limits. This is crucial for 'complex logic,' as it prevents the transaction from hitting CPU time or heap size limits that would occur if one tried to process all 100,000 records in a single synchronous transaction. Batch Apex also provides built-in state management and error handling (via Database.RaisesPlatformEvents or the finish method).

Option B (Queueable) is better suited for smaller, chained tasks; while it can handle some volume, it is not optimized for 100,000 records in the same way the QueryLocator in Batch Apex is. Option C is a syntactical impossibility as executeBatch only accepts Batchable classes. Option D (Schedulable) is used for timing, but the actual heavy lifting for 100,000 records would still need to be handed off to a Batch class to avoid timeout errors. Therefore, Database.Batchable is the optimal tool for large-scale data seeding.

Question 3

Question Type: MultipleChoice

A company accepts orders for customers in their ERP system that must be integrated into Salesforce as Order__c records with a lookup field to Account. The Account object has an external ID field, ERP_Customer_ID__c. What should the integration use to create new Order__c records that will automatically be related to the correct Account?1234

Options:

A- Upsert on the Order__c object and specify the ERP_Customer_ID__c for the Account

relationship.5678
B- Upsert on the Account and specify the ERP_Customer_ID__c for the relations9hip.101112
C- Merge on the Order__c object and spe13cify the ERP_Customer_ID__c for 14the 15Account relationship.
D- Insert on the Order__c object followed by an update on the Order__c object.

Answer:

A

Explanation:

In Salesforce integration, the Upsert operation is highly powerful because it allows you to create or update records and relate them to other records using External IDs instead of Salesforce record IDs ($001...$).

When the ERP system sends an order, it may not know the 18-character Salesforce ID for the Account, but it does know the ERP_Customer_ID__c. By performing an Upsert on the Order__c object (Option A), the integration can leverage 'Foreign Key' syntax in the API payload. This essentially tells Salesforce: 'Create this Order and link its Account__c lookup field to the Account that has this specific ERP_Customer_ID__c.'

This approach is the 'Gold Standard' for integrations because:

It eliminates the need for the external system to perform a 'lookup' query in Salesforce to find an ID before creating a record.

It reduces the total number of API calls, saving on governor limits.

It handles 'idempotency,' ensuring that if the same order is sent twice, it is updated rather than duplicated.

Options B, C, and D are less efficient. Upserting the Account (Option B) doesn't create the Order. Merging (Option C) is for deduplication. The Insert-then-Update approach (Option D) is an anti-pattern that doubles the required API calls and transaction overhead.

Question 4

Question Type: MultipleChoice

A developer wrote the following method to find all the test accounts in the org:

Java

public static Account[] searchTestAccounts() {

List searchList = [FIND 'test' IN ALL FIELDS RETURNING Account(Name)];

return (Account[]) searchList[0];

}

However, the test method below fails.

Java

@isTest

public static void testSearchTestAccounts() {

Account a = new Account(name='test');

insert a;

Account [] accounts = TestAccountFinder.searchTestAccounts();

System.assert(accounts.size() == 1);

}

What should be used to fix this failing test?

Options:

A- @isTest(SeeAllData=true) to access org data for the test9
B- Test.loadData to set up expected data10
C- Test.setFixedSearchResults() method to set up expected data11
D- @testSetup method to set up expected data12

Answer:

C

Explanation:

14

In Sales15force, SOSL (Search) queries behave differently than SOQL (Database) queries during unit tests. Because the search index is not updated in real-time during a test transaction, a SOSL FIND statement will consistently return an empty list, even if records matching the criteria were just inserted within the same test method. This leads to the assertion failure seen in the developer's code.

To overcome this, Salesforce provides the Test.setFixedSearchResults() method (Option C). This method allows a developer to define exactly which record IDs should be returned by any SOSL

query executed during the test. When Test.setFixedSearchResults(list_of_ids) is called, the platform bypasses the actual search engine and returns the specified records. This ensures that the test remains deterministic and can verify the logic that processes the search results.

Option A (SeeAllData=true) is a poor practice and doesn't guarantee the search index will be ready. Option B (Test.loadData) and Option D (@testSetup) are for data creation but do not solve the underlying SOSL indexing limitation. Using setFixedSearchResults is the standard, documented way to test SOSL functionality in the Apex environment.

==========

Question 5

Question Type: MultipleChoice

Which statement is true regarding savepoints?

Options:

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

Explanation:

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.

==========

Question 6

Question Type: MultipleChoice

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?

Options:

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

Explanation:

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.

Question 7

Question Type: MultipleChoice

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?

Options:

A- Implement the Batchable interface.
B- Use platform events.
C- Use the @future annotation.
D- Implement the Queueable interface.

Answer:

A

Explanation:

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.

Question 8

Question Type: MultipleChoice

A developer is building a complex commission calculation engine in Apex that is called from an Opportunity trigger. During QA it was reported that the calculations are incorrect. The developer has representative test data and passing test methods in their developer sandbox. Which three tools or techniques could the developer use to execute the code and pause it at key lines to visually inspect values of various Apex variables?

Options:

A- Apex Interactive Debugger
B- Apex Replay Debugger
C- Workbench
D- Developer Console
E- Breakpoints

Answer:

A, B, E

Explanation:

To perform traditional 'step-through' debugging in Salesforce---where you can pause execution and inspect the state of variables in real-time or near-real-time---developers use the Salesforce extensions for VS Code.

The Apex Replay Debugger (Option B) is a free tool that allows developers to 'replay' a transaction using a debug log. By setting Breakpoints (Option E) in their code and then executing the logic (or running a test), the developer can generate a log file. The Replay Debugger then parses this log, allowing the developer to step through the code line-by-line in VS Code as if it were happening live1.2

The Apex Interactive Debugger (Option A) is a specia3lized tool (typically requiring a paid add-on for sandboxes) that allows for live, real-time debugging on a sandbox or scratch org. Unlike the Replay Debugger, it pauses the actual execution on the Salesforce server, giving the developer a live view of the environment.

The Developer Console (Option D) and Workbench (Option C) provide log viewing and query capabilities, but they do not support the ability to 'pause' execution at specific lines for visual inspection in the manner of a traditional IDE debugger. Therefore, the combination of a debugger tool and the use of breakpoints is the standard requirement for this debugging pattern.

Question 9

Question Type: MultipleChoice

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?

Options:

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

Explanation:

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.

==========

Question 10

Question Type: MultipleChoice

A developer notices the execution of all the test methods in a class takes a long time to run, due to the initial setup of all the test data that is needed to perform the tests. What should the developer do to speed up test execution?

Options:

A- Define a method that creates test data and annotate with @createData.
B- Reduce the amount of test methods in the class.12
C- Define a method that creates test data and annotate with @testSetup.
D- Ensure proper usage of test data factory in all test methods.34

Answer:

C

Explanation:

Comprehens7ive and Detailed 150 8to 250 words of

In Salesforce Apex testing, the @testSetup annotation is a powerful tool designed to improve test performance and reduce code redundancy. When a method is marked with @testSetup, it executes once before any of the individual test methods in the class run. The data created in this method is then persisted for the entire class.

The primary performance benefit comes from how Salesforce handles this data: the platform creates a 'snapshot' of the database state after the setup method finishes. For every subsequent test method in the class, the system simply rolls back the database to this snapshot rather than re-executing the data creation logic. This significantly reduces the time spent on DML operations, which are often the most time-consuming part of a test suite.

Option D (Data Factories) is a best practice for code reuse, but if called inside every test method, it still results in redundant DML operations. Options A and B are incorrect; @createData is not a valid Salesforce annotation, and reducing the number of tests sacrifices code coverage and quality. By using @testSetup, a developer ensures that heavy data initialization happens only once, leading to faster execution cycles and more efficient resource usage during deployments.

==========

Question 11

Question Type: MultipleChoice

A company wants to incorporate a third-party web service to set the Address fields when an Account is inserted, if they have not already been set. What is the optimal way to achieve this?

Options:

A- Create a Before Save Flow, execute a Queueable job from it, and make a callout from the Queueable job.
B- Create an Apex class, execute a Batch Apex job from it, and make a callout from the Batch Apex job.
C- Create an Apex trigger, execute a Queueable job from it, and make a callout from the Queueable job.
D- Create an Apex class, execute a Future method from it, and make a callout from the Future method.

Answer:

C

Explanation:

The requirement is to perform a callout when a record is inserted. Because Salesforce prohibits synchronous callouts after DML operations in the same transaction, this must be handled asynchronously.

Queueable Apex (Option C) is the optimal choice for this integration. When an Account is inserted, a trigger captures the ID and enqueues a Queueable job. Queueable is superior to Future methods (Option D) because it supports complex data types (not just primitives), allows for job chaining, and provides a Job ID that can be used for monitoring via AsyncApexJob.

Before-Save Flows (Option A) cannot directly perform callouts, and while they can trigger asynchronous paths, the programmatic control offered by a Trigger-to-Queueable pattern is more robust for complex third-party integrations. Batch Apex (Option B) is designed for bulk processing of existing records and is 'overkill' for a real-time, record-by-record trigger requirement. By using a Trigger with Queueable, the developer ensures the address validation happens nearly in real-time without blocking the user's initial save transaction or hitting concurrent request limits.

Question 12

Question Type: MultipleChoice

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 ahList = AccountHistoryManager.getAccountHistory(a);

System.assert(ahList.size() > 0);

}

What should be done to make this test pass?

Options:

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

Explanation:

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

Question 13

Question Type: MultipleChoice

A developer used custom settings to store some configuration data that changes occasionally. However, tests are now failing in some of the sandboxes that were recently refreshed. What should be done to eliminate this issue going forward?

Options:

A- Set the setting type on the custom setting to List.
B- Replace custom settings with static resources.1
C- Replace custom settings with custom metadata.2
D- Set the setting type on the custom setting to Hierarchy.3

Answer:

C

Explanation:

5

The iss6ue described is a direct result of how Salesforce treats data during sandbox refreshes and unit testing. Custom Settings (both List and Hierarchy) store their information as data records. By default, sandbox refreshes do not include data records from Production unless it is a Full Sandbox. Furthermore, Apex unit tests are isolated from organization data; they cannot

'see' Custom Setting records unless the records are explicitly created within the test or the @isTest(SeeAllData=true) annotation is used (which is a poor practice).

Custom Metadata (Option C) solves this by treating configuration records as metadata rather than data. Because they are metadata, the records are automatically included in every sandbox refresh and are deployable via Change Sets or the Salesforce CLI. Most importantly, Custom Metadata is visible to Apex unit tests without needing to create setup data or bypass test isolation. This ensures that configuration-dependent logic remains consistent across all environments and that tests pass reliably after a refresh. Replacing Custom Settings with Custom Metadata is the modern Salesforce standard for environment-independent configuration management.