COURSE PROGRESS: 0% COMPLETE

Salesforce Platform Developer I – Practice Test #5

Results

0 of 40 Questions answered correctly

Your time: 00:00:02

You have reached 0 of 40 point(s), (0%)

Restart Quiz View Questions

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	32
33	34	35	36	37	38	39	40								

When designing an Apex trigger for a custom object that updates a related object's records, which considerations should be taken into account regarding Apex transactions, the save order of execution, and the potential for recursion and/or cascading effects?

- [ ] Ensure that the trigger checks for recursive calls by using a static variable, to prevent the trigger from executing more than once in the same transaction when updating related records.

Correct answer

Correct al

Live Agent

- [ ] Configure the trigger to use a custom setting or a static variable as a flag to prevent it from executing the logic again if it's already run for the same record in the same transaction.
- [ ] Place all logic directly inside the trigger, including queries and updates to related records, to ensure that all operations are executed in a single transaction for consistency.
- [ ] Use after update triggers exclusively to modify related records, since this guarantees that all changes to the original record are committed before related records are updated.

Incorrect

Ensure that the trigger checks for recursive calls by using a static variable, to prevent the trigger from executing more than once in the same transaction when updating related records. -> Correct. Utilizing a static variable to prevent recursive trigger execution is a best practice for managing the save order of execution and avoiding unintended recursion or cascading effects.

Configure the trigger to use a custom setting or a static variable as a flag to prevent it from executing the logic again if it’s already run for the same record in the same transaction. -> Correct. Using a custom setting or static variable as a flag to prevent re-execution of the trigger for the same record in a transaction is a recommended practice to manage recursion and ensure efficient and safe operations.

Place all logic directly inside the trigger, including queries and updates to related records, to ensure that all operations are executed in a single transaction for consistency. -> Incorrect. While maintaining transactional consistency is important, placing all logic directly in the trigger can lead to maintenance issues and risks of hitting governor limits. It’s better to separate concerns and manage recursion carefully.

Use after update triggers exclusively to modify related records, since this guarantees that all changes to the original record are committed before related records are updated. -> Incorrect. While using after update triggers is a common practice for updating related records, it does not inherently protect against recursion or cascading effects without proper checks and balances.

In developing a comprehensive Salesforce application, a developer integrates various components such as triggers, Apex classes, custom controllers, and automated processes. To ensure the highest code quality and application reliability, the developer must adopt best practices for testing and deployment. Which approach should the developer take to effectively write and execute tests for these components, considering the need to cover a wide range of scenarios and data conditions?

Implement test methods that cover only the most common use cases to reduce the complexity and time required for testing.

Rely exclusively on real-time data in the production environment to test all components for the most accurate results.

Utilize the Test.loadData() method to load complex test data from static resources, ensuring diverse testing scenarios for components.

Correct answer

Use the @isTest(SeeAllData=true) annotation sparingly to access existing org data for specific scenarios where creating test data is impractical.

Incorrect

Utilize the Test.loadData() method to load complex test data from static resources, ensuring diverse testing scenarios for components. -> Correct. Test.loadData() is a powerful tool for loading complex data sets from static resources, allowing developers to test components under various scenarios without manually creating extensive test data, which is efficient for testing complex relationships and data models.

Rely exclusively on real-time data in the production environment to test all components for the most accurate results. -> Incorrect. Testing directly in production with real-time data is risky and can affect live data and users. Salesforce best practices recommend creating mock data within test classes for a controlled and isolated testing environment.

Implement test methods that cover only the most common use cases to reduce the complexity and time required for testing. -> Incorrect. While focusing on common use cases is important, comprehensive testing should also include edge cases and negative scenarios to ensure robustness and prevent future errors.

Use the @isTest(SeeAllData=true) annotation sparingly to access existing org data for specific scenarios where creating test data is impractical. -> Incorrect. Although using @isTest(SeeAllData=true) can be necessary in rare cases, the correct approach is to create test data within test classes whenever possible to ensure tests are not dependent on org data.

In Apex, how would you correctly declare a constant that represents the maximum number of attempts allowed for a login process, ensuring it cannot be modified after its initial assignment?

public final int MAX_LOGIN_ATTEMPTS = 5; Correct answer

private int maxLoginAttempts = 5;

public static int MAX_LOGIN_ATTEMPTS = 5;

final int MAX_LOGIN_ATTEMPTS = 5;

Incorrect

public final int MAX_LOGIN_ATTEMPTS = 5; -> Correct. This declaration correctly defines a constant using the final keyword, which prevents modification after initial assignment, and public to make it accessible outside the class.

private int maxLoginAttempts = 5; -> Incorrect. This declaration makes maxLoginAttempts a modifiable instance variable, not a constant, due to the lack of the final keyword and it being private limits its accessibility.

public static int MAX_LOGIN_ATTEMPTS = 5; -> Incorrect. While static allows the variable to be accessed without an instance, omitting final means it's not a constant and can be modified.

final int MAX_LOGIN_ATTEMPTS = 5; -> Incorrect. This declaration correctly makes MAX_LOGIN_ATTEMPTS a constant, but without an access modifier like public, its accessibility is default, limiting use outside its package.

Given a requirement to implement a complex business logic that involves multiple objects and needs to be executed both before and after records are saved, what is the best practice for structuring your Apex classes and triggers?

Avoid using Apex and seek to implement the requirement entirely through Workflow Rules and Process Builder for better performance.

Create separate triggers for each operation (before insert, after insert, etc.) on each object to simplify the logic.

Design a single, generic trigger per object that calls specific handler classes based on the context of the trigger operation (before insert, after insert, etc.).

Correct answer

Implement all business logic directly in a single trigger for each object to minimize the number of classes and triggers.

Incorrect

Design a single, generic trigger per object that calls specific handler classes based on the context of the trigger operation (before insert, after insert, etc.). -> Correct. This approach adheres to the best practice of having one trigger per object and using handler classes to organize and manage the business logic efficiently, making the solution scalable and maintainable.

Implement all business logic directly in a single trigger for each object to minimize the number of classes and triggers. -> Incorrect. Implementing all logic directly in a single trigger can lead to complex, hard-to-maintain code and may exceed governor limits.

Create separate triggers for each operation (before insert, after insert, etc.) on each object to simplify the logic. -> Incorrect. Creating separate triggers for each operation can lead to conflicts, unpredictable behavior due to execution order, and difficulties in maintaining the code.

Avoid using Apex and seek to implement the requirement entirely through Workflow Rules and Process Builder for better performance. -> Incorrect. While declarative tools are powerful, Apex provides the capability to implement complex business logic that cannot be achieved declaratively, especially when dealing with multiple objects or complex logic.

In an Apex class that integrates with an external service, you're tasked with implementing exception handling to gracefully manage service failures and alert the system when specific conditions are met. If the service response indicates an unauthorized access attempt, a custom exception should be thrown to enforce security protocols. How should this functionality be properly implemented in Apex?

Use a try-catch block to catch generic exceptions and log each occurrence.

Create a custom exception class and use it directly within a try block to handle unauthorized access.

Use a try-catch block, and within the catch block, throw a custom exception if the error message indicates an unauthorized access attempt.

Correct answer

Implement a try-catch-finally block, throwing the custom exception in the finally block if unauthorized access is detected.

Incorrect

Use a try-catch block, and within the catch block, throw a custom exception if the error message indicates an unauthorized access attempt. -> Correct. This method properly implements exception handling by using a try-catch block to catch exceptions that occur during the external service call. By inspecting the exception or error message within the catch block and throwing a custom exception specifically for unauthorized access attempts, it aligns with the requirement to enforce security protocols for this specific condition.

Use a try-catch block to catch generic exceptions and log each occurrence. -> Incorrect. Catching and logging generic exceptions can help with monitoring errors but fails to address the requirement to throw a custom exception for unauthorized access attempts, missing the specificity needed for security-related issues.

Create a custom exception class and use it directly within a try block to handle unauthorized access. -> Incorrect. Although creating a custom exception class is on the right track, using it directly within a try block without catching any exceptions does not implement the proper exception handling mechanism required to manage various runtime exceptions effectively.

Implement a try-catch-finally block, throwing the custom exception in the finally block if unauthorized access is detected. -> Incorrect. The finally block is not the correct place to throw exceptions based on conditional logic, as it executes after the try and catch blocks regardless of whether an exception was thrown, leading to potential misuse of the custom exception handling.

In a scenario where a Salesforce developer is tasked with creating a data model for a new customer support application within Salesforce, which two actions are considered best practices in determining, creating, and accessing the appropriate data model, including objects, fields, relationships, and external IDs? Select two correct answers.

- [ ] Design a lookup relationship between the Customer Support Case and Product objects instead of a master-detail relationship to allow cases to exist independently of specific products.

Correct answer

- [ ] Utilize a many-to-many relationship via a junction object to relate Customers to Products, enabling the tracking of product issues reported by multiple customers.

- [ ] Use external ID fields on the Customer object to enable more efficient integration with external systems and streamline data synchronization processes. Correct answer

- [ ] Establish a master-detail relationship between the Customer Support Case and Customer objects to ensure direct ownership and facilitate roll-up summaries of case metrics.

- [ ] Implement custom objects with text fields for all unique identifiers to maximize flexibility in record identification and integration scenarios.

Incorrect

Use external ID fields on the Customer object to enable more efficient integration with external systems and streamline data synchronization processes. -> Correct. External ID fields are specifically designed for integration use cases, making them ideal for matching and upserting records during integration operations, thus improving efficiency and accuracy in data synchronization.

Design a lookup relationship between the Customer Support Case and Product objects instead of a master-detail relationship to allow cases to exist independently of specific products. -> Correct. Lookup relationships offer flexibility by allowing related records to exist independently. This is crucial in customer support scenarios where a case might not always be directly associated with a specific product, or where the product association might change.

Establish a master-detail relationship between the Customer Support Case and Customer objects to ensure direct ownership and facilitate roll-up summaries of case metrics. -> Incorrect. While master-detail relationships provide strong linkage and enable roll-up summaries, they might be too restrictive for customer support cases where cases may need to exist even if the customer record is deleted or not directly associated.

Implement custom objects with text fields for all unique identifiers to maximize flexibil record identification and integration scenarios. -> Incorrect. While text fields offer

flexibility, using them for unique identifiers without the external ID attribute can lead to inefficiencies in data management and integration tasks, making it harder to ensure uniqueness and match records accurately.

Utilize a many-to-many relationship via a junction object to relate Customers to Products, enabling the tracking of product issues reported by multiple customers. -> Incorrect. Creating a many-to-many relationship through a junction object is a best practice for tracking complex associations, such as multiple customers reporting issues for the same product. However, it was not selected as one of the top two best practices in the context of this question's focus on integration and relationship flexibility.

Which statement accurately describes the Lightning Component framework, its benefits, and the types of content that can be contained in a Lightning web component?

This framework is primarily for external website development, supporting only static HTML content without the ability to integrate with Salesforce data.

The framework is a UI development toolkit for developing web apps for the Salesforce Lightning Platform, allowing for responsive applications across devices. It can contain HTML, JavaScript, and CSS. Correct answer

The Lightning Component framework is designed exclusively for mobile applications, offering high performance and flexibility, and can only contain HTML content.

Lightning Component framework is a server-side framework that can only execute Apex code and is not suitable for client-side interactions or UI development.

Incorrect

The framework is a UI development toolkit for developing web apps for the Salesforce Lightning Platform, allowing for responsive applications across devices. It can contain HTML, JavaScript, and CSS. -> Correct. The Lightning Component framework is part of Salesforce Lightning platform, providing a modern framework for building dynamic we

applications for desktop and mobile devices. It benefits from being highly responsive, scalable, and efficient for developers, allowing the inclusion of HTML for structure, JavaScript for client-side logic, and CSS for styling, enabling the creation of rich, interactive user interfaces that can seamlessly integrate with Salesforce data.

The Lightning Component framework is designed exclusively for mobile applications, offering high performance and flexibility, and can only contain HTML content. -> Incorrect. The framework is not exclusively for mobile applications and can contain more than just HTML content.

Lightning Component framework is a server-side framework that can only execute Apex code and is not suitable for client-side interactions or UI development. -> Incorrect. The Lightning Component framework is client-side, designed for creating user interfaces, and does support client-side interactions.

This framework is primarily for external website development, supporting only static HTML content without the ability to integrate with Salesforce data. -> Incorrect. The framework is specifically designed for developing applications on the Salesforce Lightning Platform, not external websites, and can interact with Salesforce data.

When integrating Apex code with Salesforce user interface components, such as Lightning Components, Flows, and Next Best Actions, which approach best aligns with Salesforce development best practices for efficient and maintainable code?

Using static Apex methods with hard-coded logic specific to each UI component type to ensure customized behavior for Lightning Components, Flows, and Next Best Actions.

Directly embedding SOQL queries within Lightning Component controller methods to fetch data specific to the component's needs.

Implementing Apex classes that expose generic, reusable APIs for data operations, which can be invoked from any UI component type, including Lightning Components, Flows, and Next Best Actions.

Correct answer

Avoiding the use of Apex classes for UI interactions and relying solely on built-in configuration options and formula fields to implement business logic.

Incorrect

Implementing Apex classes that expose generic, reusable APIs for data operations, which can be invoked from any UI component type, including Lightning Components, Flows, and Next Best Actions. -> Correct. Creating generic, reusable Apex APIs promotes code reusability and maintainability, allowing different UI components to interact with backend logic efficiently without duplicating code.

Directly embedding SOQL queries within Lightning Component controller methods to fetch data specific to the component's needs. -> Incorrect. Directly embedding SOQL queries in Lightning Component controllers is not a best practice. It's better to encapsulate data access in Apex classes, providing a separation of concerns and reusability.

Using static Apex methods with hard-coded logic specific to each UI component type to ensure customized behavior for Lightning Components, Flows, and Next Best Actions. -> Incorrect. Using static methods with hard-coded logic reduces flexibility and reusability. Apex should be designed to be as dynamic and reusable as possible, with customization handled through parameters rather than hard coding.

Avoiding the use of Apex classes for UI interactions and relying solely on built-in configuration options and formula fields to implement business logic. -> Incorrect. While leveraging built-in configurations and formula fields is encouraged for simple logic, Apex is necessary for complex business logic that cannot be achieved through configurations alone.

Which of the following declarative process automation features in Salesforce allows for the execution of actions in a specified order and can perform actions not only when records are created or updated but also upon meeting certain criteria at any point in time?

Flow

Process Builder

Approval Processes

Workflow Rules

Incorrect

Flow -> Correct. Salesforce Flow is a powerful automation tool that allows for the execution of complex logic in a specified order. Unlike Workflow Rules and Process Builder, Flows can be triggered by various events, not limited to record creation or updates. They can execute actions based on record changes, user interactions, or at specific times, offering a highly flexible solution for automating complex business processes.

Workflow Rules -> Incorrect. Workflow Rules can automate simple actions when records are created or updated but do not offer the ability to execute actions in a specified sequence or trigger based on complex criteria beyond initial creation or update events.

Process Builder -> Incorrect. While Process Builder allows for more complex automation than Workflow Rules, including executing actions in a certain order when records are created or updated, it is not as versatile as Flow in handling various triggering events or executing without a record change.

Approval Processes -> Incorrect. Approval Processes are designed specifically for record approval workflows. They do not offer general automation capabilities and are focused on guiding records through a series of approval steps.

A developer is tasked with creating a complex sales tracking application in Salesforce. The application requires a custom object, Sales_Record__c, which stores information about each sale, including Sale_Amount__c (currency), Sale_Date__c (date), and Customer_Region__c (text). The developer needs to ensure that whenever a new Sales_Record__c is created or updated, a calculation is performed to apply a discount based on the Customer_Region__c. The discount logic is as follows:
“North America”: 10% discount
“Europe”: 15% discount
“Asia”: 5% discount
Other regions: No discount
Additionally, the application must store the discounted sale amount in a new field, Discounted_Amount__c, and ensure that the logic accounts for changes in the Customer_Region__c field. To implement the above scenario, which approach should the developer take?

Use an Apex Trigger on Sales_Record__c — Correct answer

Use a Workflow Rule with Field Updates

Use Formula Fields on Sales_Record__c

Use a Process Builder

Incorrect

Use an Apex Trigger on Sales_Record__c -> Correct. An Apex Trigger on Sales_Record__c allows for the execution of complex logic based on the record’s fields. It can perform calculations and apply the necessary discounts based on the Customer_Region__c efficiently.

Use a Workflow Rule with Field Updates -> Incorrect. While Workflow Rules can update fields based on certain criteria, they do not support complex logic that varies by region requires calculation of percentages.

Use a Process Builder -> Incorrect. Although Process Builder can handle more complex logic than Workflow Rules and can perform field updates, it is not the most efficient way to handle calculations and region-based logic due to its limited ability to execute complex logic without calling Apex code.

Use Formula Fields on Sales_Record__c -> Incorrect. Formula fields calculate values dynamically based on other fields' values but cannot perform actions like creating or updating records, making them unsuitable for requirements that involve storing calculated results.

In a Salesforce development project, you are tasked with implementing a custom business logic that requires iterating over a list of custom objects, Invoice__c, to calculate the total amount due for each customer based on specific conditions. Each Invoice__c object has two fields: Amount__c (Currency) and Status__c (String) which can have values 'Pending', 'Paid', or 'Cancelled'. You need to calculate the total due amount only for 'Pending' invoices. Which Apex control flow statement and logic correctly accomplishes this task?

Use a for-each loop without any conditional checks, assuming all invoices are 'Pending' by default.

Use a do-while loop with a switch statement on Status__c to add up amounts, including a case for 'Pending'.

Use a while loop without any conditional statements to sum up the Amount__c for all invoices, regardless of their status.

Use a for loop and an if-else statement to iterate over the invoices and sum up the Amount__c for each 'Pending' invoice.

Correct answer

Incorrect

Use a for loop and an if-else statement to iterate over the invoices and sum up the Amount__c for each 'Pending' invoice. -> Correct. A for loop correctly iterates over eac

Live Agent

invoice, and the if-else statement allows for conditionally adding only 'Pending' invoices to the total due amount.

Use a while loop without any conditional statements to sum up the Amount__c for all invoices, regardless of their status. -> Incorrect. A while loop could iterate over the list, but without conditionals, it doesn't correctly sum only the 'Pending' invoices.

Use a do-while loop with a switch statement on Status__c to add up amounts, including a case for 'Pending'. -> Incorrect. While a do-while loop will ensure at least one iteration and a switch can handle multiple conditions, this approach is less efficient for iterating over collections in Apex.

Use a for-each loop without any conditional checks, assuming all invoices are 'Pending' by default. -> Incorrect. Assuming all invoices are 'Pending' ignores the need to verify each invoice's status, leading to inaccurate total calculations.

When writing unit tests for a custom Apex class in Salesforce, which statement best ensures that the test data is isolated from the organization's actual data and does not affect the organization's data limits?

Implement Database.insertImmediate() for test data creation to bypass triggers and workflows.

Use SeeAllData=true in the test class to access real data for more comprehensive testing.

Directly insert test data in test methods without any specific annotation or method.

Apply @isTest(SeeAllData=false) at the class level, the default behavior, to ensure test data isolation. Correct answer

Incorrect

Apply @isTest(SeeAllData=false) at the class level, the default behavior, to ensure test data isolation. -> Correct. By default, the @isTest(SeeAllData=false) annotation isolates test data from actual organizational data, ensuring tests do not affect org data limits and are isolated.

Use SeeAllData=true in the test class to access real data for more comprehensive testing. -> Incorrect. Using SeeAllData=true allows tests to access all data in the organization, which can lead to tests that are not isolated and might affect actual data.

Implement Database.insertImmediate() for test data creation to bypass triggers and workflows. -> Incorrect. Database.insertImmediate() does not exist in Salesforce Apex as a method for test data creation.

Directly insert test data in test methods without any specific annotation or method. -> Incorrect. Directly inserting test data without any specific mechanism does not ensure isolation from the organization's actual data.

When developing an Apex trigger for a Salesforce custom object, it is crucial to understand the relationship between Apex transactions, the save order of execution, and the potential for recursion and/or cascading. Given this context, which statement best describes how to manage these aspects to ensure efficient and bug-free code?

Design triggers to execute all logic directly without using helper classes or methods, as this simplifies the save order of execution and reduces the risk of recursion.

Utilize static variables within helper classes to track execution and prevent the same trigger from executing more than once in the context of the same transaction, effectively managing recursion.

Correct answer

Implement the trigger logic without checking for recursion since Salesforce automatically handles trigger recursion and ensures that each trigger fires only once per transaction.

Live Agent

Focus on the order in which records are updated in the trigger to manage recursion, assuming that updating records in a specific sequence prevents recursive trigger execution.

Incorrect

Utilize static variables within helper classes to track execution and prevent the same trigger from executing more than once in the context of the same transaction, effectively managing recursion. -> Correct. This is a common and effective approach to managing recursion in Apex triggers. Static variables can be used to flag whether a trigger has already run, preventing it from executing again in the same transaction.

Implement the trigger logic without checking for recursion since Salesforce automatically handles trigger recursion and ensures that each trigger fires only once per transaction. -> Incorrect. Salesforce does not automatically prevent trigger recursion. Developers must implement their own mechanisms to prevent recursion to ensure triggers do not re-execute in the same transaction.

Design triggers to execute all logic directly without using helper classes or methods, as this simplifies the save order of execution and reduces the risk of recursion. -> Incorrect. Directly coding all logic in triggers without helper classes does not mitigate the risk of recursion and complicates the understanding of the save order of execution. Helper classes can help organize logic and manage recursion effectively.

Focus on the order in which records are updated in the trigger to manage recursion, assuming that updating records in a specific sequence prevents recursive trigger execution. -> Incorrect. While the order of record updates can influence trigger behavior, it does not inherently prevent recursion. Specific mechanisms, such as static flags, are required to manage recursion effectively.

What are best practices for writing and executing tests for triggers, controllers, classes, flows, and processes in Salesforce, especially regarding the use of various sources of test data? Select two.

- [ ] Ensure test methods verify the behavior of automation, such as validation rules and workflow rules, under different data conditions. Correct answer
- [ ] Rely on existing data in the target environment by setting SeeAllData=true for all test classes to simplify test setup.
- [ ] Always use SeeAllData=true to ensure that tests are as realistic as possible, mimicking the exact production environment.
- [ ] Utilize the Test.loadData() method to load test data from static resource files for complex data models. Correct answer

Incorrect

Utilize the Test.loadData() method to load test data from static resource files for complex data models. -> Correct. The Test.loadData() method allows for the loading of test data from static resources, enabling the testing of complex data models without manually creating each record in test methods. This approach promotes test data reuse and consistency across test classes.

Ensure test methods verify the behavior of automation, such as validation rules and workflow rules, under different data conditions. -> Correct. It's important for test methods to verify that triggers, classes, and automation like validation rules and workflow rules behave correctly under various conditions. This includes testing with different data inputs to ensure robustness and reliability.

Rely on existing data in the target environment by setting SeeAllData=true for all test classes to simplify test setup. -> Incorrect. Setting SeeAllData=true is generally discouraged because it makes tests dependent on the data present in the environment leading to less reliable and more brittle tests that can fail if the data changes.

Always use SeeAllData=true to ensure that tests are as realistic as possible, mimicking the exact production environment. -> Incorrect. While using real data might seem beneficial for realism, it can lead to unpredictable test results and dependencies on specific data setups. The best practice is to create test data within test methods.

In a complex Salesforce application, a developer is tasked with automating a process that updates Contact records based on changes to related Account records. The developer decides to use an Apex trigger on the Account object to implement this logic. Considering the relationship between Apex transactions, the save order of execution, and the potential for recursion and/or cascading, what is the most important consideration the developer should keep in mind to ensure efficient and error-free execution?

Utilize the @future annotation to delay updates to Contact records

Use workflow rules instead of an Apex trigger for updates

Implement a static variable in the Apex trigger to prevent recursion Correct answer

Ensure the Apex trigger is set to execute only after an Account record is deleted

Incorrect

Implement a static variable in the Apex trigger to prevent recursion -> Correct. Utilizing a static variable within the trigger can help in managing and preventing recursion by checking if the trigger has already run in the current transaction, thus avoiding potential issues with cascading updates.

Ensure the Apex trigger is set to execute only after an Account record is deleted -> Incorrect. While managing trigger execution timing is important, focusing on deletion doesn't address the relationship between Apex transactions, save order, and prevention of recursion or cascading in this scenario.

Utilize the @future annotation to delay updates to Contact records -> Incorrect. While @future can manage asynchronous operations, it's not specifically about managing recursion or ensuring the correct order of execution within Apex transactions related to Account and Contact records.

Use workflow rules instead of an Apex trigger for updates -> Incorrect. Workflow rules might simplify some automation but cannot handle complex logic that requires the consideration of Apex transactions, save order, and recursion or cascading effects as efficiently as an Apex trigger can.

Which two scenarios are best suited for using a Full Sandbox? Select two answers.

- [ ] Performance Load Testing — Correct answer
- [ ] Continuous Integration Testing
- [ ] User Acceptance Testing (UAT) — Correct answer
- [ ] Debugging Live Data Issues
- [ ] Application Security Testing

Incorrect

User Acceptance Testing (UAT) -> Correct. A Full Sandbox provides a complete replica of the production environment, making it ideal for UAT to ensure the application behaves as expected with real data.

Performance Load Testing -> Correct. Because Full Sandboxes copy the entire data set from production, they are suitable for performance load testing to measure how the application behaves under heavy data volumes and user loads.

Debugging Live Data Issues -> Incorrect. While Full Sandboxes contain all production data, it's generally recommended to debug live data issues in a more controlled environment to avoid impacting the full dataset.

Continuous Integration Testing -> Incorrect. Continuous integration testing typically requires a smaller, more controlled environment to rapidly test and integrate code changes, making a Developer Sandbox or a Developer Pro Sandbox more appropriate.

Application Security Testing -> Incorrect. While application security testing can be performed in a Full Sandbox, it does not require the full data set that a Full Sandbox offers. A Partial Copy Sandbox or even a Developer Sandbox with mock data could suffice for many security testing scenarios.

A Salesforce developer is designing a data model for a custom application that manages real estate properties. The application needs to track properties, property listings, and brokers. Each property can have multiple listings over time, but each listing is managed by a single broker. Additionally, the application must integrate with an external system that uses unique property identifiers. To ensure efficient data management and integration, while adhering to Salesforce best practices, which of the following data model designs is most appropriate?

Correct answer

Create a custom object for properties, listings, and brokers. Use a master-detail relationship between properties and listings, and a lookup relationship between listings and brokers. Assign an external ID field on the property object.

Implement all three as custom objects but connect them using only master-detail relationships, including a master-detail from listings to brokers.

Use a single custom object for both properties and listings, with a text field to differentiate between the two. Utilize a lookup relationship for brokers connected to this single object.

Create separate custom objects for properties, listings, and brokers with lookup relationships between all three objects. Use a unique text field on properties as an informal external ID.

Incorrect

Create a custom object for properties, listings, and brokers. Use a master-detail relationship between properties and listings, and a lookup relationship between listings and brokers. Assign an external ID field on the property object. -> Correct. This design effectively captures the required relationships and hierarchy between properties, listings, and brokers. The master-detail relationship enforces data integrity and roll-up summaries between properties and listings, while the lookup relationship allows for flexibility with brokers. The external ID facilitates integration with external systems.

Use a single custom object for both properties and listings, with a text field to differentiate between the two. Utilize a lookup relationship for brokers connected to this single object. -> Incorrect. This design oversimplifies the model, leading to potential data management issues and inefficiencies. It fails to utilize relationships to accurately represent the data hierarchy and complicates the tracking of properties and their listings over time.

Create separate custom objects for properties, listings, and brokers with lookup relationships between all three objects. Use a unique text field on properties as an informal external ID. -> Incorrect. While separate objects are used, relying solely on lookup relationships misses the opportunity to enforce data integrity and hierarchy between properties and listings. Additionally, using a unique text field as an informal external ID is less reliable and efficient than using Salesforce's designated external ID field type.

Implement all three as custom objects but connect them using only master-detail relationships, including a master-detail from listings to brokers. -> Incorrect. Using a master-detail relationship between listings and brokers unnecessarily restricts the model, as it would not allow a broker to exist without an associated listing, which does not align with the given requirements.

To guarantee that a field in a Salesforce object captures a specific, pre-defined category of information without repetition and is always populated, which two field properties should be set?

- [x] Unique — Correct answer
- [ ] External ID
- [ ] Validation Rule
- [x] Required — Correct answer

Incorrect

Unique -> Correct. Marking a field as unique ensures that each record will have a distinct value for this field, preventing duplicate entries.

Required -> Correct. Setting a field to required ensures that it cannot be left empty, guaranteeing that data is entered every time a record is created or edited.

External ID -> Incorrect. An external ID is used for records integration and can be unique, but its primary purpose is not to ensure data uniqueness or mandatory entry in the context of general data capture.

Validation Rule -> Incorrect. Validation rules enforce data quality and specific conditions but do not directly ensure that a field is always populated or unique; they define conditions that must be met for the record to be saved.

A Salesforce team is planning to deploy a complex update involving new custom objects, Apex triggers, Lightning components, and several configuration changes. The update needs thorough testing to ensure it does not negatively impact the existing functionality and performance in the production environment. What process and environments should the team use to ensure a smooth and successful deployment?

Use Change Sets to promote changes from a Full Copy Sandbox to Production after comprehensive testing. Correct answer

Directly deploy from a Developer Sandbox to Production to expedite the deployment process.

Utilize Unmanaged Packages to move changes from development to production, bypassing sandbox testing.

Conduct all testing in a Production environment using live data to ensure accuracy.

Incorrect

Use Change Sets to promote changes from a Full Copy Sandbox to Production after comprehensive testing. -> Correct. Deploying changes using Change Sets from a Full Copy Sandbox, which mirrors production data and configurations closely, allows for comprehensive testing and minimizes risks to production users and data.

Directly deploy from a Developer Sandbox to Production to expedite the deployment process. -> Incorrect. Deploying directly from a Developer Sandbox to Production without thorough testing in environments that mirror production closely, such as Full Sandboxes or Staging environments, risks introducing untested changes to end-users.

Utilize Unmanaged Packages to move changes from development to production, bypassing sandbox testing. -> Incorrect. While Unmanaged Packages can be used for deployment, bypassing sandbox testing, especially for complex updates, ignores best practices for quality assurance and risk mitigation.

Conduct all testing in a Production environment using live data to ensure accuracy. -> Incorrect. Testing in a Production environment can disrupt business operations and expose live data to unintended changes or breaches, which is against best practices and security protocols.

In a complex Salesforce application, a developer is tasked with implementing a custom user interface that enables users to dynamically filter a list of opportunities based on multiple criteria and then perform bulk actions, such as updating or deleting selected records. This interface must be seamlessly integrated within the Salesforce mobile app as well as the desktop version. Considering these requirements, which of the following approaches should the developer choose?

Use Salesforce Flow to build the user interface.

Create a custom Aura Component.

Develop a Visualforce page with embedded Apex controllers.

Implement a Lightning Web Component (LWC). Correct answer

Incorrect

Implement a Lightning Web Component (LWC). -> Correct. LWCs offer modern web standards-based development that enables creating efficient, dynamic, and mobile-responsive user interfaces. They can interact with Apex to perform bulk actions and can be easily integrated into Salesforce mobile and desktop environments.

Use Salesforce Flow to build the user interface. -> Incorrect. While Flow can automate complex business processes and handle bulk actions, it does not offer the flexibility needed for creating highly customizable and dynamic user interfaces for displaying and filtering records.

Develop a Visualforce page with embedded Apex controllers. -> Incorrect. Visualforce pages can create custom user interfaces and integrate Apex for logic; however, they might not provide the optimal user experience on mobile devices compared to newer technologies.

Create a custom Aura Component. -> Incorrect. Although Aura Components can be used for this purpose and are mobile-responsive, LWCs are the more modern solution recommended by Salesforce for new development due to their performance benefits and alignment with web standards.

In an Apex class, a developer needs to write a query to fetch the name and email of all contacts related to a specific Account ID. Which SOQL query correctly accomplishes this task, ensuring optimal performance and best practices?

SELECT Name, Email FROM Contact WHERE AccountId IN :listOfAccountIds

SELECT Name, Email FROM Contact WHERE AccountId = :accountId Correct answer

SELECT Name, Email FROM Contacts WHERE AccountId = :accountId

SELECT Contact.Name, Contact.Email FROM Account WHERE Id = :accountId

Incorrect

SELECT Name, Email FROM Contact WHERE AccountId = :accountId -> Correct. This query is efficient and follows best practices by directly querying the Contact object with a specific Account ID using a bind variable for optimal performance.

SELECT Name, Email FROM Contacts WHERE AccountId = :accountId -> Incorrect. The query syntax is incorrect; the object name should be 'Contact' not 'Contacts', and it lacks proper object relationship querying.

SELECT Name, Email FROM Contact WHERE AccountId IN :listOfAccountIds -> Incorrect. While this query is syntactically correct, it is designed for querying multiple accounts, not a specific Account ID.

SELECT Contact.Name, Contact.Email FROM Account WHERE Id = :accountId -> Incorrect. This query attempts to query from the Account object down to Contacts, which is an incorrect approach given the scenario. It would not execute as intended because the relationship query syntax is incorrect for accessing contacts directly from an Account query in this context.

In Apex programming, when declaring variables, constants, methods, and using modifiers and Apex interfaces, which two of the following statements are accurate?

- [ ] Apex does not support method overloading within a class.
- [ ] Instance methods require the static modifier to be accessible without creating an object of the class.
- [ ] The final keyword is used to declare constants which value cannot be modified after initialization. Correct answer
- [ ] Access modifiers like public and private define the visibility of variables and methods to other classes. Correct answer
- [ ] Apex interfaces can contain concrete method implementations.

Incorrect

The final keyword is used to declare constants which value cannot be modified after initialization. -> Correct. The final keyword is indeed used to declare constants in Apex, indicating that once a variable is assigned a value, it cannot be altered. This is useful for defining values that should remain constant throughout the execution of a program.

Live Agent

Access modifiers like public and private define the visibility of variables and methods to other classes. -> Correct. Access modifiers such as public and private control the accessibility of classes, methods, and variables in Apex. Public allows access from any class, whereas private restricts access to the class in which they are declared.

Instance methods require the static modifier to be accessible without creating an object of the class. -> Incorrect. Instance methods do not require the static modifier; it's the static methods that can be accessed without instantiating the class. The static modifier is used for class-level fields and methods that can be accessed without creating an instance of the class.

Apex interfaces can contain concrete method implementations. -> Incorrect. Apex interfaces cannot contain concrete method implementations. Interfaces only declare method signatures. The implementing classes are responsible for providing the concrete implementation of these methods.

Apex does not support method overloading within a class. -> Incorrect. Apex does support method overloading, allowing multiple methods in the same class to have the same name but different parameters. This enables different implementations or actions based on the method signature called.

In which scenario would a system administrator be unable to delete a custom Apex class from Salesforce?

The Apex class is mentioned in a Lightning Component.

The Apex class is part of a process builder process.

The Apex class is scheduled for future execution.

The Apex class is referenced in a Visualforce page.

Correct answer

Incorrect

The Apex class is referenced in a Visualforce page. -> Correct. Apex classes that are directly referenced by Visualforce pages cannot be deleted until those references are removed because the Visualforce page would fail to compile.

The Apex class is scheduled for future execution. -> Incorrect. While you might need to unschedule the class before deletion, scheduling alone does not prevent deletion.

The Apex class is part of a process builder process. -> Incorrect. While Apex classes can be invoked by Process Builder, the existence of such a reference does not inherently prevent the class's deletion. Proper checks and balances should be in place before deletion.

The Apex class is mentioned in a Lightning Component. -> Incorrect. Similar to Visualforce, but the platform allows for the deletion with proper handling in Lightning Components. However, it's best practice to ensure no dependencies exist before deletion.

A custom object record triggers a process builder flow upon creation, which includes an immediate action to update a related record and a scheduled action to send an email reminder to the record owner 7 days after the record's creation date. If the record is modified 3 days after creation and no longer meets the initial criteria of the process builder, what happens to the scheduled email reminder?

The scheduled email reminder is cancelled, and no email is sent. Correct answer

The scheduled email reminder is rescheduled for 7 days after the record modification date.

The process builder recalculates the scheduled send date based on the modification but still sends the email 7 days after the original creation date.

The email reminder is immediately sent upon modification of the record.

Incorrect

The scheduled email reminder is cancelled, and no email is sent. -> Correct. If the record is modified in such a way that it no longer meets the criteria for the process, any scheduled actions that have not yet occurred are cancelled.

The scheduled email reminder is rescheduled for 7 days after the record modification date. -> Incorrect. Scheduled actions are based on the criteria being met at the time of record creation or modification that triggers the process. If criteria are no longer met, actions are not rescheduled but cancelled.

The email reminder is immediately sent upon modification of the record. -> Incorrect. The modification of the record leading to it no longer meeting the process criteria cancels the scheduled action, rather than triggering immediate actions.

The process builder recalculates the scheduled send date based on the modification but still sends the email 7 days after the original creation date. -> Incorrect. Process builder does not recalculate scheduled actions based on record modifications that cause the record to no longer meet the triggering criteria; instead, such actions are cancelled.

When developing a user interface for a Salesforce application, which statement best describes the use of the Lightning Component Framework, its benefits, and the capabilities of Lightning Web Components (LWC)?

The Lightning Component Framework is exclusively server-side, rendering all components on the server before delivering them to the client.

LWCs are limited to utilizing only JavaScript and HTML, excluding the use of external CSS files for styling purposes.

Correct answer

The Lightning Component Framework provides a modern UI development approach using standard web technologies like HTML, JavaScript, and CSS, enabling the creation of reusable components that encapsulate functionality and design, and LWCs can contain HTML, JavaScript, CSS, and use Salesforce data and metadata through Apex controllers or wire services.

LWCs support direct embedding of SOQL queries within the component's HTML template for dynamic data retrieval.

Incorrect

The Lightning Component Framework provides a modern UI development approach using standard web technologies like HTML, JavaScript, and CSS, enabling the creation of reusable components that encapsulate functionality and design, and LWCs can contain HTML, JavaScript, CSS, and use Salesforce data and metadata through Apex controllers or wire services. -> Correct. This statement accurately encapsulates the essence of the Lightning Component Framework and LWCs. It highlights the modern web standards-based approach, the encapsulation of functionality and design into reusable components, and the ability to interact with Salesforce data and metadata, showcasing the benefits and capabilities of using LWCs in Salesforce application development.

LWCs support direct embedding of SOQL queries within the component's HTML template for dynamic data retrieval. -> Incorrect. SOQL queries cannot be directly embedded within an LWC's HTML template. Data retrieval in LWC is accomplished through Apex methods or wire services, not direct query embedding in the template.

LWCs are limited to utilizing only JavaScript and HTML, excluding the use of external CSS files for styling purposes. -> Incorrect. LWCs allow the use of external CSS files for styling, in addition to supporting CSS styles defined within the component, providing comprehensive styling options.

The Lightning Component Framework is exclusively server-side, rendering all components on the server before delivering them to the client. -> Incorrect. The Lightning Component Framework includes both client-side and server-side aspects. LWCs are primarily client

side, leveraging modern web standards for rendering and interaction, but can interact with the server for data operations.

In a scenario where a developer needs to quickly update a Visualforce page and its associated Apex controller to fix a critical issue in a production environment, which Salesforce development tool offers the most direct and efficient path for making and deploying these urgent changes?

Developer Console for immediate editing and deployment of Visualforce pages and Apex controllers. Correct answer

Salesforce DX for version-controlled development and team collaboration.

Salesforce CLI for bulk data operations and complex script executions.

Workbench for SOQL queries and direct data manipulation.

Incorrect

Developer Console for immediate editing and deployment of Visualforce pages and Apex controllers. -> Correct. Developer Console allows for direct editing of Visualforce pages and Apex controllers in the browser, providing a quick way to make and deploy changes in production environments.

Salesforce CLI for bulk data operations and complex script executions. -> Incorrect. While Salesforce CLI is powerful for automation and script execution, it's not the primary tool for direct, urgent edits to Visualforce pages and Apex controllers.

Salesforce DX for version-controlled development and team collaboration. -> Incorrect. Salesforce DX excels in collaborative development and version control but may not be the quickest option for urgent, direct changes in production due to its focus on source-driven development workflows.

Workbench for SOQL queries and direct data manipulation. -> Incorrect. Workbench is a versatile tool for data and metadata operations but does not specialize in the direct editing or quick deployment of Visualforce pages and Apex controllers.

In Salesforce, what are two fundamental practices a developer should follow when writing Apex code? Select two.

- [ ] Hardcode IDs in Apex classes to ensure consistent reference to specific records.
- [ ] Follow the Bulkify principle to ensure code properly handles multiple records at a time. Correct answer
- [ ] Utilize SOQL queries within for loops to handle large data sets efficiently.
- [ ] Store business logic in triggers instead of classes to simplify code management.
- [ ] Apply the principle of least privilege by using the 'with sharing' keyword where appropriate. Correct answer

Incorrect

Follow the Bulkify principle to ensure code properly handles multiple records at a time. -> Correct. Bulkification ensures that the Apex code efficiently processes large numbers of records by minimizing the number of SOQL queries and DML operations, thus avoiding governor limits.

Apply the principle of least privilege by using the ‘with sharing’ keyword where appropriate. -> Correct. Using the ‘with sharing’ keyword enforces record-level security and sharing rules in Apex classes, ensuring that users can only access data they’re authorized to view.

Hardcode IDs in Apex classes to ensure consistent reference to specific records. -> Incorrect. Hardcoding IDs is considered a bad practice because it makes the code less flexible and can lead to errors in different environments where record IDs change.

Utilize SOQL queries within for loops to handle large data sets efficiently. -> Incorrect. Placing SOQL queries inside for loops can quickly hit governor limits on the number of SOQL queries allowed, making it an inefficient practice.

Store business logic in triggers instead of classes to simplify code management. -> Incorrect. Storing business logic directly in triggers is not recommended. It's better to call classes from triggers for better organization and reusability of the code.

In a scenario requiring the integration and usage of custom user interface components within Salesforce, including Lightning Components, Flow, and Visualforce, which method ensures the most effective and seamless implementation?

Directly call Apex classes from JavaScript within Lightning Components without handling any callback functions to streamline data operations and user interactions.

For optimal performance and compatibility, build custom UI components using only HTML and CSS, avoiding JavaScript to simplify development within the Salesforce ecosystem.

Embed Lightning Components into Visualforce pages using Lightning Out, allowing for the reuse of custom Lightning components within Visualforce, ensuring a unified experience across Salesforce UI technologies.

Correct answer

Use Aura Components to wrap Visualforce pages and embed them within Lightning Experience and Salesforce Mobile App, leveraging Aura's event-driven architecture for interaction.

Incorrect

Embed Lightning Components into Visualforce pages using Lightning Out, allowing for the reuse of custom Lightning components within Visualforce, ensuring a unified experience across Salesforce UI technologies. -> Correct. Embedding Lightning Components into Visualforce pages using Lightning Out enables developers to leverage the modern,

component-based architecture of Lightning Web Components (or Aura Components) within the traditional Visualforce page framework. This approach allows for the seamless integration of custom UI components across Salesforce's UI technologies, offering a consistent user experience while taking advantage of the performance and modularity of Lightning Components.

Directly call Apex classes from JavaScript within Lightning Components without handling any callback functions to streamline data operations and user interactions. -> Incorrect. Directly calling Apex classes from JavaScript without handling callback functions can lead to unresponsive UIs and data handling errors, as asynchronous operations are a core part of modern web development.

For optimal performance and compatibility, build custom UI components using only HTML and CSS, avoiding JavaScript to simplify development within the Salesforce ecosystem. -> Incorrect. Avoiding JavaScript limits the functionality and interactivity of custom UI components within Salesforce, where JavaScript plays a crucial role in creating dynamic user experiences.

Use Aura Components to wrap Visualforce pages and embed them within Lightning Experience and Salesforce Mobile App, leveraging Aura's event-driven architecture for interaction. -> Incorrect. While Aura Components can indeed embed Visualforce pages, this description oversimplifies the complexities and potential limitations of integrating these technologies in a way that might not always lead to the most effective or seamless implementation.

Given a scenario where you need to iterate over a list of Account records and perform a specific action only on those accounts where the AnnualRevenue is greater than $1 million, which Apex control flow statement would you use?

Switch Statement

While Loop

Correct answer

For Loop

If-Else Statement

Incorrect

For Loop -> Correct. A for loop is ideal for iterating over a list of records and checking each one's AnnualRevenue attribute to perform actions on those meeting the criteria.

If-Else Statement -> Incorrect. While if-else statements are used for conditional logic, they're not the best choice for iterating over lists.

While Loop -> Incorrect. A while loop could theoretically be used, but it's less efficient and more complex for this task than a for loop, especially when dealing with lists.

Switch Statement -> Incorrect. A switch statement is used for branching logic based on matching a value to multiple cases, not for iterating over lists.

When developing a complex application with multiple Lightning Web Components (LWC), what is the most effective strategy for leveraging events to facilitate component communication and interaction?

Use the @api decorator to expose public methods on a component, allowing it to emit events to parent components without using the standard event dispatch mechanism.

Employ non-bubbling events for all inter-component communications to ensure that events do not propagate beyond their intended scope, using composed: false in every event.

Broadcast a global event from a child component to update all instances of a sibling component directly, bypassing the parent component.

Dispatch custom events with minimal payload only when necessary, and listen for these events in parent components to handle data updates or UI changes.

Incorrect

Dispatch custom events with minimal payload only when necessary, and listen for these events in parent components to handle data updates or UI changes. -> Correct. This practice aligns with best practices by using custom events judiciously, focusing on efficiency and specificity. Minimal payloads ensure that only necessary data is communicated, reducing overhead and adhering to LWC's component-based architecture's performance and modularity principles.

Broadcast a global event from a child component to update all instances of a sibling component directly, bypassing the parent component. -> Incorrect. LWC does not support global events in the same manner as application events in Aura Components. Communication should be managed through parent components or using a pub-sub model for loosely coupled components, not by directly targeting sibling components.

Use the @api decorator to expose public methods on a component, allowing it to emit events to parent components without using the standard event dispatch mechanism. -> Incorrect. The @api decorator is used to expose public properties and methods, but it is not directly related to event handling. Events are dispatched using the dispatchEvent method, not through @api exposed methods.

Employ non-bubbling events for all inter-component communications to ensure that events do not propagate beyond their intended scope, using composed: false in every event. -> Incorrect. While non-bubbling events (composed: false) are useful for containing events within a specific component or a closely related group of components, not all inter-component communications should be restricted in this manner. The use of bubbling or non-bubbling events should be based on the specific communication needs of the components.

Which of the following is a best practice for writing test classes in Apex?

Make HTTP callouts in test methods to test integrations with external systems directly.

Hard-code test data in test methods to ensure consistency across test runs.

Use the @isTest(SeeAllData=true) annotation to access all organizational data for comprehensive testing.

Utilize the Test.startTest() and Test.stopTest() methods to define test execution limits. Correct answer

Incorrect

Utilize the Test.startTest() and Test.stopTest() methods to define test execution limits. -> Correct. These methods are used to demarcate a section of the test method to run with its own governor limits, allowing for more robust and isolated testing of Apex code.

Hard-code test data in test methods to ensure consistency across test runs. -> Incorrect. Hard-coding test data is not recommended because it can lead to tests that fail when the data schema or business logic changes. Salesforce recommends using test setup methods to create test data dynamically.

Use the @isTest(SeeAllData=true) annotation to access all organizational data for comprehensive testing. -> Incorrect. Using @isTest(SeeAllData=true) is generally not recommended because it can lead to tests that are dependent on org data, making them less reliable and more difficult to maintain. Salesforce encourages creating test data within test classes.

Make HTTP callouts in test methods to test integrations with external systems directly. -> Incorrect. Making HTTP callouts directly from test methods is not possible because tests are run in a simulated environment. Salesforce recommends using mock callouts for testing external integrations.

Tech Innovations wants to automate the process of updating their Salesforce contacts with information from their internal HR system whenever an employee's details change. What is the best approach for a developer to implement this integration?

Implement a REST API integration using Apex callouts to the HR system. Correct answer

Employ Workflow Rules to send email alerts to administrators for manual updates.

Configure Process Builder to automatically export CSV files to the HR system.

Use Salesforce-to-Salesforce integration to connect with the HR system.

Incorrect

Implement a REST API integration using Apex callouts to the HR system. -> Correct. This method allows Salesforce to programmatically request updates from the HR system, ensuring real-time synchronization.

Use Salesforce-to-Salesforce integration to connect with the HR system. -> Incorrect. Salesforce-to-Salesforce integration is specifically designed for linking two Salesforce orgs, not for connecting Salesforce with external systems.

Configure Process Builder to automatically export CSV files to the HR system. -> Incorrect. While Process Builder can automate many tasks within Salesforce, it does not natively support exporting CSV files to external systems.

Employ Workflow Rules to send email alerts to administrators for manual updates. -> Incorrect. This solution still relies on manual processes and does not directly automate the synchronization of data between Salesforce and external systems.

In a given scenario where you need to query a list of Contacts associated with Accounts in the healthcare industry and then update their status to “Active” in Apex, which combination of SOSL, SOQL, and DML statements should you use?

- [ ] Use SOSL to find Accounts in the healthcare industry, then SOQL to query Contacts related to those Accounts, and finally, a DML update statement to set their status to "Active".
- [ ] Use SOSL to search for all Contacts and Accounts simultaneously, then filter Contacts related to healthcare Accounts in Apex, and use a DML update statement on the filtered Contacts. Correct answer
- [ ] Use DML update directly on Contacts filtered by Account industry through Apex logic without a query.
- [ ] Use SOQL to query Contacts directly based on Account industry criteria, then use a DML update statement to set their status to "Active". Correct answer
- [ ] First, apply a DML insert operation to clone the Contacts, then use SOQL to query these cloned Contacts and update their status to "Active".

Incorrect

Use SOQL to query Contacts directly based on Account industry criteria, then use a DML update statement to set their status to “Active”. -> Correct. This approach efficiently retrieves Contacts based on Account criteria in a single SOQL query and then updates their status directly.

Use SOSL to search for all Contacts and Accounts simultaneously, then filter Contacts related to healthcare Accounts in Apex, and use a DML update statement on the filtered Contacts. -> Correct. This option is also correct but it’s less efficient and not as direct as using SOQL for specific queries related to relationship criteria.

Use SOSL to find Accounts in the healthcare industry, then SOQL to query Contacts re to those Accounts, and finally, a DML update statement to set their status to “Active”. -

Incorrect. SOSL is not designed for specific relationship queries like finding Contacts related to specific Accounts; SOQL is more appropriate for this task.

Use DML update directly on Contacts filtered by Account industry through Apex logic without a query. -> Incorrect. DML operations cannot filter records; a query is needed to identify which records to update.

First, apply a DML insert operation to clone the Contacts, then use SOQL to query these cloned Contacts and update their status to “Active”. -> Incorrect. This approach unnecessarily complicates the task with cloning operations and does not efficiently target the actual requirement.

When implementing exception handling in Apex, especially in scenarios requiring custom exceptions, what approach should a developer take to ensure robust and maintainable error management?

Use Database.insert(records, false) and handle all exceptions in a single catch block, regardless of the operation's nature.

Implement custom exceptions by extending the built-in Exception class and throw these exceptions to handle specific error conditions.

Correct answer

Rely solely on system-defined exceptions and avoid custom exceptions to reduce complexity in error handling.

Use a generic try-catch block for all exceptions and log the error message to a custom object for all exceptions.

Incorrect

Implement custom exceptions by extending the built-in Exception class and throw these exceptions to handle specific error conditions. -> Correct. Creating custom exceptions extending Salesforce’s Exception class allows developers to define error conditions tha

specific to their application's logic, facilitating more precise and meaningful error handling. This approach enables targeted catch blocks that can handle specific custom exceptions differently, improving the robustness and clarity of the error handling mechanism.

Use a generic try-catch block for all exceptions and log the error message to a custom object for all exceptions. -> Incorrect. While logging errors to a custom object can be part of a comprehensive error handling strategy, using a generic catch block for all exceptions does not allow for differentiated handling based on the exception type, which can be critical for resolving specific issues effectively.

Rely solely on system-defined exceptions and avoid custom exceptions to reduce complexity in error handling. -> Incorrect. Relying only on system-defined exceptions limits the ability to handle custom error scenarios that are specific to your business logic. Custom exceptions are essential for representing and handling errors that fall outside the range of predefined Salesforce exceptions.

Use Database.insert(records, false) and handle all exceptions in a single catch block, regardless of the operation's nature. -> Incorrect. Although using Database.insert(records, false) allows for partial success of DML operations, handling all exceptions in a single catch block does not provide the granularity needed for comprehensive error analysis and handling. Different types of operations and exceptions require different handling strategies.

Given a scenario where a Salesforce admin needs to aggregate data from a child object to a parent object in a one-to-many relationship, which approach best aligns with Salesforce best practices and considerations such as governor limits, formula fields, and roll-up summaries?

Implement a roll-up summary field on the parent object to automatically calculate totals from the child records. Correct answer

Develop a Lightning Component to manually calculate and update the aggregated data on the parent object.

Create a Visualforce page to display aggregated data on the parent object without storing the data.

Use a complex Apex trigger to calculate and update the parent record every time a child record is created or updated.

Incorrect

Implement a roll-up summary field on the parent object to automatically calculate totals from the child records. -> Correct. Roll-up summary fields are a declarative feature designed for this exact use case, allowing for efficient aggregation of data without code, thus adhering to best practices and avoiding governor limit issues.

Use a complex Apex trigger to calculate and update the parent record every time a child record is created or updated. -> Incorrect. While Apex triggers can achieve this, it's not the best practice when declarative options like roll-up summaries are available, as triggers may introduce unnecessary complexity and governor limit concerns.

Create a Visualforce page to display aggregated data on the parent object without storing the data. -> Incorrect. This approach displays aggregated data but doesn't store it on the parent object, which may not fulfill the requirement if data storage on the parent is needed.

Develop a Lightning Component to manually calculate and update the aggregated data on the parent object. -> Incorrect. Similar to Apex triggers, developing a Lightning Component for this task introduces unnecessary complexity when simpler declarative options exist.

Given a scenario where you need to find all Contact records with the last name 'Smith' and then update their MailingCity to 'New York', which combination of SOSL, SOQL, and DML statements in Apex would accomplish this task correctly?

Contact[] contacts = [FIND : 'Smith' IN LASTNAME FIELDS RETURNING Contact][0]; for(Contact c : contacts) { c.MailingCity = 'New York'; } update contacts;

List contacts = [SELECT MailingCity FROM Contact WHERE LastName LIKE 'Smith']; for(Contact c : contacts) { c.MailingCity = 'New York'; } Database.update(contacts);

List contacts = [FIND 'Smith' IN ALL FIELDS RETURNING Contact]; for(Contact c : contacts) { c.MailingCity = 'New York'; } update contacts;

Contact[] contacts = [SELECT Id FROM Contact WHERE LastName = 'Smith']; for(Contact c : contacts) { c.MailingCity = 'New York'; } update contacts;

Correct answer

Incorrect

Contact[] contacts = [SELECT Id FROM Contact WHERE LastName = ‘Smith’];
for(Contact c : contacts) {
c.MailingCity = ‘New York’;
}
update contacts;

This uses SOQL to query the correct records, iterates over them to set the new MailingCity, and then uses DML to update the records in the database.

Incorrect answers:

List contacts = [FIND ‘Smith’ IN ALL FIELDS RETURNING Contact];
for(Contact c : contacts) {
c.MailingCity = ‘New York’;
}
update contacts;

SOSL syntax is incorrect for finding specific fields; SOSL is used for text searches across multiple objects.

List contacts = [SELECT MailingCity FROM Contact WHERE LastName LIKE ‘Smith’];
for(Contact c : contacts) {
c.MailingCity = ‘New York’;
}
Database.update(contacts);

The query only retrieves the MailingCity field, which is insufficient for the DML update operation as the Id field is also needed.

```
Contact[] contacts = [FIND : 'Smith' IN LASTNAME FIELDS RETURNING Contact][0];
for(Contact c : contacts) {
c.MailingCity = 'New York';
}
update contacts;
```

This is an attempt to mix SOSL search syntax in a SOQL query, which is not valid. SOSL and SOQL have distinct syntaxes and purposes.

Given a scenario where you need to implement a custom user interface component to display a dynamic list of records that updates in real-time based on user input, which two of the following options would be the most appropriate to use?

- [ ] Flow with Screen Elements
- [ ] Apex Class with SOQL Queries embedded in Visualforce
- [ ] Aura Component with an Apex Controller — Correct answer
- [ ] Lightning Web Component (LWC) utilizing Lightning Data Service (LDS) — Correct answer
- [ ] Visualforce Page with Standard Controllers

Incorrect

Lightning Web Component (LWC) utilizing Lightning Data Service (LDS) -> Correct. LWC with LDS allows for the creation of responsive UIs that can react to data changes in real-time, making it suitable for dynamic list updates based on user interactions.

Aura Component with an Apex Controller -> Correct. Aura Components, especially when used with an Apex Controller, can be designed to update the UI in real-time in response to user actions, providing a rich interactive experience.

Visualforce Page with Standard Controllers -> Incorrect. Standard Controllers are primarily used for simple CRUD operations and do not support real-time updates based on user input without additional custom Apex code and AJAX.

Apex Class with SOQL Queries embedded in Visualforce -> Incorrect. While this method can display data, it does not inherently support real-time updates without implementing complex logic for polling or refreshing, which is not as efficient as using more modern frameworks.

Flow with Screen Elements -> Incorrect. Flow is powerful for guiding users through a process but is not the best tool for creating dynamic, real-time updating lists based on user input without custom development.

Global Enterprises has set the organization-wide default for Leads to public read-only to enhance collaboration across the sales team. However, the marketing team reports that their lead views are cluttered with leads that are not relevant to their current campaigns, making it challenging to prioritize their efforts effectively. How can the System Administrator make it easier for the marketing team to focus on their relevant leads?

Implement customized list views for campaign-specific leads. Correct answer

Customize lead page layouts to highlight campaign-related fields.

Refine the sharing rules for lead visibility among teams.

Modify the lead access settings on user profiles.

Incorrect

Implement customized list views for campaign-specific leads. -> Correct. Customized list views can filter leads based on specific criteria, such as campaign association, enabling the marketing team to focus on leads that are most relevant to their efforts.

Refine the sharing rules for lead visibility among teams. -> Incorrect. Sharing rules are designed to increase record visibility, not to organize or filter records for easier access based on campaign relevance.

Customize lead page layouts to highlight campaign-related fields. -> Incorrect. While customizing page layouts can make certain information more prominent, it does not address the issue of filtering leads for easier access.

Modify the lead access settings on user profiles. -> Incorrect. Changing access settings on user profiles affects record access permissions but does not help in organizing or filtering leads to manage visibility based on campaign relevance.

Given a scenario where a developer needs to create a Visualforce page that displays a custom object's data with dynamic search capabilities, allowing users to filter results based on specific criteria, which two solutions should the developer use to implement this functionality effectively, utilizing Visualforce pages and the appropriate controllers or extensions?

- [ ] Use HTML5 data attributes within the Visualforce page to store and filter data without server-side processing.
- [ ] Embed external JavaScript libraries directly into the Visualforce page to handle data fetching and filtering on the client side.
- [ ] Create a custom controller that includes methods for fetching and filtering data based on user input.
- [ ] Utilize a standard controller with Visualforce page bindings for dynamic search filters.

Correct answer

Correct al

Live Agent

- [ ] Implement an extension to the standard controller that adds filtering logic to the existing data retrieval process.

Incorrect

Create a custom controller that includes methods for fetching and filtering data based on user input. -> Correct. A custom controller allows for complete control over the data fetching and filtering logic, enabling the developer to implement complex search functionality tailored to specific requirements.

Implement an extension to the standard controller that adds filtering logic to the existing data retrieval process. -> Correct. Extensions augment the functionality of standard controllers, making them suitable for adding custom logic like dynamic filtering while still leveraging the built-in features of the standard controller.

Utilize a standard controller with Visualforce page bindings for dynamic search filters. -> Incorrect. Standard controllers are limited in customization and may not support complex dynamic search functionality directly. They are better suited for straightforward data display and manipulation without extensive custom logic.

Embed external JavaScript libraries directly into the Visualforce page to handle data fetching and filtering on the client side. -> Incorrect. While external JavaScript libraries can enhance the UI, relying on them for data operations bypasses the server-side logic and security features provided by Apex controllers and is not recommended for manipulating Salesforce data.

Use HTML5 data attributes within the Visualforce page to store and filter data without server-side processing. -> Incorrect. HTML5 data attributes can store data on the client side, but relying solely on them for data manipulation and filtering lacks scalability and security, missing out on the server-side processing capabilities of Apex.

A Salesforce developer is tasked with designing a robust user interface for a custom application within the Salesforce ecosystem. To leverage the full potential of the platform, the developer decides to use the Lightning Component framework. Which of the following best describes the Lightning Component framework, its benefits, and the types of content that can be contained in a Lightning web component?

It is a UI-centric development framework that enhances user and developer productivity, offers benefits such as component reusability and event-driven architecture, and can contain HTML, JavaScript, and CSS content. Correct answer

A framework that exclusively supports the development of backend processes, offering benefits such as automated testing and continuous integration, and contains only Apex classes and triggers.

It is a server-side framework that improves data handling operations, primarily benefits from accelerated server processing, and exclusively contains Apex and SOQL content.

The Lightning Component framework is a UI development framework specifically designed for the Salesforce mobile app, offering benefits such as mobile optimization and the ability to contain only HTML and JavaScript content.

Incorrect

It is a UI-centric development framework that enhances user and developer productivity, offers benefits such as component reusability and event-driven architecture, and can contain HTML, JavaScript, and CSS content. -> Correct. This accurately describes the Lightning Component framework, highlighting its UI focus, key benefits, and support for standard web technologies.

The Lightning Component framework is a UI development framework specifically designed for the Salesforce mobile app, offering benefits such as mobile optimization and the ability to contain only HTML and JavaScript content. -> Incorrect. While the Lightning Component framework is mobile-optimized, it is not exclusive to the Salesforce mobile app and can contain more than just HTML and JavaScript.

It is a server-side framework that improves data handling operations, primarily benefits from accelerated server processing, and exclusively contains Apex and SOQL content. -> Incorrect. The Lightning Component framework is client-side, focusing on UI development, not server-side operations, and does not exclusively contain Apex and SOQL content.

A framework that exclusively supports the development of backend processes, offering benefits such as automated testing and continuous integration, and contains only Apex classes and triggers. -> Incorrect. The Lightning Component framework is focused on UI development rather than backend processes and does not exclusively contain Apex classes and triggers.