Boost up Your Certification Score

Guidewire
InsuranceSuite-Developer
Associate Certification - InsuranceSuite Developer - Mammoth Proctored Exam

For More Information – Visit link below:

https://www.examsboost.com/

Product Version

✓ Up to Date products, reliable and verified.
✓ Questions and Answers in PDF Format.

Latest Version: 6.1

Question: 1

An insurer doing business globally wants to use a validation expression to verify that a contact's postal code is a real postal code for the country specified in the contact's address.
A developer has created a method with the signature validatePostalCode(anAddress: Address): boolean, which returns true if and only if the postal code is valid.
What would be the correct validation expression?

A. validatePostalCode(anAddress) == true
B. validatePoslalCode(anAddress) == null
C. validatePostalCode(anAddress)
D. validatePostalCode(anAddress) ? null: false

Answer: D

Explanation:
In Guidewire InsuranceSuite configuration, Validation Expressions (found in the Data Model within entity properties or specialized validation files) follow a specific logic: they must return null if the data is valid, and a non-null value (typically a String representing the error message) if the data is invalid. This is a common point of confusion for developers used to standard Boolean logic where "true" means valid.
In this scenario, the helper method validatePostalCode returns a Boolean (true for valid, false for invalid). Because Guidewire expects a null result for success, a direct call to a Boolean method is insufficient.
Option A and C are incorrect because they evaluate to a Boolean value (true or false). If the method returns true, the validation engine receives true instead of null, which it incorrectly interprets as a validation failure.
Option B is syntactically incorrect as it compares a Boolean to null.
Option D uses the ternary operator to map the Boolean result to the expected Guidewire logic. If validatePostalCode is true (valid), the expression returns null, which the system treats as "no error found." If the method is false (invalid), it returns a non-null value (in this exam wording, false). Since false is not null, the Guidewire validation engine identifies this as a failure and prevents the data from being committed or advancing to the next stage.
While best practices in a production environment suggest returning a descriptive String for the user (e.g., "Invalid postal code for this country"), for the purposes of the developer exam, the logic focuses on the Null/Non-Null requirement. This mechanism ensures that developers understand how the Guidewire validation framework triggers errors based on the presence of a return value rather than a Boolean state.

Visit us at: https://www.examsboost.com/test/insurancesuite-developer

Question: 2

You need to retrieve Claim entity instances created after a specific date. Which methods ensure that the filtering is performed in the database for optimal performance?

A. Retrieve all claims and filter the collection in Gosu memory using the where ( ) method.
B. Retrieve claims using a query and then filter the results collection using the filterwhere method.
C. Use the filter () .where () methods on the query object to filter the records by their creation date.
D. Use the compare method on the query object to filter claim records by their creation date.
E. Use the where method on the query object to filter claim records by their creation date.

Answer: D

Explanation:
In Guidewire InsuranceSuite development, performance is heavily dependent on how data is retrieved from the relational database. When dealing with potentially large datasets, such as the Claim entity, it is critical to perform filtering at the database level (via SQL WHERE clauses) rather than at the application level (in Gosu memory).
The Guidewire Query API provides the primary mechanism for constructing these database-level filters. When a developer creates a query object (e.g., gw.api.database.Query.make(Claim)), they must use specific methods to define the criteria that will be translated into a SQL query. The compare() method is the standard approach for adding these constraints. It allows the developer to specify the property (such as CreateTime), the comparison operator (such as GreaterThan), and the value (the specific date). Because the compare() method is called directly on the Query object before the query is executed, the filtering happens within the database engine.
In contrast, methods like where() or filter() used on a collection or a QueryBuilder result (Option A, C, and E) often trigger the execution of the query first, fetching all records into the Gosu application server's memory, and then discarding the ones that don't match. This "in-memory filtering" leads to severe performance degradation, high memory consumption, and potential "Out of Memory" errors. Option D correctly utilizes the Query API's ability to refine the result set at the source. Understanding the lifecycle of a query—from construction using compare() to execution—is a fundamental skill for any Guidewire developer to ensure the application remains scalable and responsive under high data volumes.

Question: 3

A customer needs the ability to categorize claims based on business needs. Which actions below follow best practices? (Choose two)

A. Define ClaimCategory_Ext as an extension of an existing claim Typelist.

B. Add a ClaimCategory_Ext Typekey to the Claim entity
C. Create a .ttx file for ClaimCategory_Ext in the Extensions\Typelist folder
D. Add a 'foreignkey' to the ClaimCategory_Ext typelist that references the Claim entity
E. Name the Typelist ClaimCategory without an _Ext suffix.
F. Create a .tti file for ClaimCategory_Ext in the Extensions\Typelist folder

Answer: B, F

Explanation:
When extending the Guidewire Data Model to meet specific business requirements, such as categorizing a Claim, developers must follow strict metadata standards. The process of adding a new categorization tool involves two primary steps: defining the list of possible values (the Typelist) and then linking that list to the business entity (the Claim).
According to Guidewire best practices, when you create a new Typelist that is not part of the base configuration, you must define it using a .tti (Typelist Interface) file. This file acts as the primary definition for the new list. Per Guidewire naming conventions, custom extensions and new metadata objects should be suffixed with _Ext to clearly distinguish them from "Out of the Box" (OOTB) components. This ensures that during future upgrades, the Guidewire upgrade tools can easily identify and preserve customer-specific configurations. Therefore, creating a .tti file named ClaimCategory_Ext.tti (Option F) is the correct procedure for initializing a new list of categories.
Once the Typelist is defined, it must be associated with the Claim entity so that each claim record can hold a specific category value. This is done by adding a new field to the Claim entity. In Guidewire, a field that references a Typelist is known as a typekey. By adding a typekey field named ClaimCategory_Ext to the Claim entity (Option B) and pointing it to the newly created Typelist, the developer enables the database to store the category selection.
Options A and C are incorrect because .ttx files are used for extending existing base typelists, not for creating entirely new ones. Option E violates the naming convention, and Option D describes a foreign key relationship which is technically different from the standard typekey implementation used for simple categorization via Typelists.

Question: 4

An insurer wants to add a new typecode for an alternate address to a base typelist EmployeeAddress that has not been extended.

A. Following best practices, which step must a developer take to perform
B. Create an EmployeeAddress_Ext.tti file and add a new typecode
C. Open the EmployeeAddress.tti and add a new typecode alternate
D. Create an EmployeeAddress.ttx file and add a new typecode
E. Create an EmployeeAddress.tix file and add a new typecode

Answer: D

Visit us at: https://www.examsboost.com/test/insurancesuite-developer

Explanation:
In the Guidewire InsuranceSuite framework, maintaining the integrity of the base configuration is paramount for ensuring a smooth upgrade path. This is achieved through a strict "extension-only" philosophy for out-of-the-box (OOTB) components. When a developer needs to modify a base typelist—like EmployeeAddress—they must understand the distinction between .tti (Typelist Interface) files and .ttx (Typelist Extension) files.
A .tti file defines the original structure and initial typecodes of a typelist. These files are considered "base" and should never be edited directly (making Option C incorrect). If a developer were to modify the base .tti, those changes would be overwritten during the next platform update. To safely add a new typecode to an existing base typelist, Guidewire requires the creation of a .ttx file with the exact same name as the base typelist (e.g., EmployeeAddress.ttx). This extension file tells the Guidewire metadata engine to merge the new entries with the existing ones at runtime.
Furthermore, Guidewire best practices for metadata extensions require specific naming conventions to prevent future "namespace collisions." While the .ttx file itself adopts the base name, the new typecode added within that file should be suffixed with _Ext (e.g., alternate_Ext). This ensures that if Guidewire later releases a product update that adds an "alternate" code to the base EmployeeAddress typelist, the customer's custom code remains unique and does not conflict with the new base code.
Option B is incorrect because you do not create a new .tti with an _Ext suffix for an existing list. Option E is incorrect because .tix is not a valid Guidewire metadata file extension; the correct extension is .ttx. Therefore, Option D is the only choice that follows the correct file creation and naming convention protocols required by the Guidewire development lifecycle.

Question: 5

As a developer you are creating a new Gosu class for Succeed Insurance. According to the course material, which of the following statements define how you should implement logging in your new class? (Choose Two)

A. All exceptions are errors, thus they should always be logged at the error level.
B. When logging an exception, provide details about the cause of the exception. Because you are providing a detailed description there is no need to log the exception message or stack trace.
C. When logging at the debug level you should check to see if debugging in enabled first to minimize possible performance issues.
D. Providing context when logging errors is essential. However, developers should avoid excessive logging, as it can be costly to implement and maintain, and it may negatively impact performance.
E. Logging in the cloud can be provided in either a string format or JSON.
F. When logging Personal Identifiable Information (PII), developers should log the information at least at the INFO level.
G. Checking the log level before logging is usually unnecessary, as logging typically has minimal impact on performance.

Answer: C, D

Explanation:
In Guidewire development, logging is a critical tool for troubleshooting and monitoring, but it must be implemented with a focus on system performance and security. According to the Guidewire InsuranceSuite Developer guidelines, "excessive logging" is a common source of performance degradation. Developers are instructed to provide meaningful context for errors so that support teams can diagnose issues without needing to reproduce them manually. However, logging should be used judiciously; logging too much data (Option D) increases I/O overhead and can clutter logs, making it difficult to find relevant information.
A specific best practice highlighted in the course material involves the use of the Debug log level. Because debug messages often involve complex string concatenation or data retrieval that consumes CPU cycles, developers should wrap these calls in a conditional check. By using if (logger.isDebugEnabled()) (Option C), the system avoids the cost of constructing the log message entirely if the current logging level is set to a higher priority, such as INFO or WARN. This practice is essential for maintaining high throughput in a production environment where debug logging is typically disabled.
Other options provided are contrary to Guidewire standards. For instance, Personal Identifiable Information (PII) (Option F) should never be logged in plain text due to data privacy regulations (GDPR/CCPA), and logging it at an INFO level would be a major security violation. Furthermore, while exceptions should be logged, not all exceptions are errors (some are expected business logic flows), and when they are logged, the stack trace is vital for debugging (refuting Option B). Guidewire Cloud primarily standardizes on structured logging (JSON) for observability, but the fundamental developer best practices regarding performance (C and D) remain the primary focus of the Fundamentals course.

Visit us at: https://www.examsboost.com/test/insurancesuite-developer

Thank You for Trying Our Product

For More Information – Visit link below:

https://www.examsboost.com/

15 USD Discount Coupon Code:

G74JA8UF

FEATURES

- ✓ 90 Days Free Updates
- ✓ Money Back Pass Guarantee
- ✓ Instant Download or Email Attachment
- ✓ 24/7 Live Chat Support
- ✓ PDF file could be used at any Platform
- ✓ 50,000 Happy Customer