TESTPASSPORT

ÜBUNGSFRAGEN

HOCHWERTIGE LERNRESSOURCEN

Exam : **InsuranceSuite Developer**

Title : Associate Certification - InsuranceSuite Developer - Mammoth Proctored Exam

Version : DEMO

1.A ListView shows related Policies for a policyholder. When a user clicks a Policy Number in a text cell, the UI should open a Popup showing details of that specific policy. The elementName property in the row iterator is currentPolicy.
What is the correct syntax to open the popup?
A. Modify the Action property on the atomic widget to PolicyPopup.go(currentPolicy)
B. Modify the actionAvailable property on the atomic widget to PolicyPopup(currentPolicy)
C. Modify the actionAvailable property on the atomic widget to PolicyPopup.push(currentPolicy)
D. Modify the Action property on the atomic widget to PolicyPopup.push(currentPolicy)
Answer: D
Explanation:
In Guidewire PCF Configuration, navigating between different parts of the application requires a clear understanding of Location types and their corresponding Gosu methods. When a requirement specifically calls for a Popup, the developer must use the .push() method.
The .push() method is used for" modal" or" semi-modal" navigation. It places the new location (the Popup) on top of the current navigation stack, allowing the user to perform a task and then return exactly where they were when the popup is dismissed. In contrast, the .go() method (seen in Option A) is used for" terminal" navigation, which replaces the current location entirely; it is used for moving between main Pages or Location Groups. Using .go() for a popup would violate the intended UI flow and likely result in a runtime error or unexpected navigation behavior.
Furthermore, the logic to trigger this navigation must be placed in the Action property of the widget (typically a TextCell or Link). The actionAvailable property (Options B and C) is a Boolean expression used only to determine if the action is clickable (i.e., whether the link is active or grayed out based on permissions or data
state); it cannot execute the navigation itself. By specifying PolicyPopup.push(currentPolicy) in the Action property, the developer ensures that the currentPolicy object (defined by the RowIterator's elementName) is passed as a parameter to the popup, allowing it to display the correct details. This follows the standard PCF Architecture for drill-down interactions.

2.A developer wants to manually trigger a build chain in TeamCity to generate a deployable Docker image for a specific commit. According to the process described in the training, what are the key initial steps to achieve this? (Choose 2)
A. Update the application's configuration resources in Lifecycle Manager.
B. Access the Automated Builds app in Guidewire Home.
C. Access TeamCity via Guidewire Home.
D. Configure a new monitor in Datadog.
E. Select the desired build configuration and trigger the build.
Answer: C E
Explanation:
In the Guidewire Cloud (GWCP) ecosystem, the CI/CD pipeline is centrally managed through Guidewire Home, which serves as the primary portal for developers to access cloud-native tools. To generate a deployable Docker image, a developer must interact with the build server, which is TeamCity. The first critical step is navigating to TeamCity via the link or tile provided in Guidewire Home (often labeled under" Automated Builds"). This ensures the developer is authenticated within the secure Guidewire Cloud environment.

Once inside TeamCity, the developer must locate the specific Build Configuration associated with the project and branch they intend to build. Because a Docker image is specific to a commit, the developer must ensure they are triggering the correct" build chain." A build chain in Guidewire Cloud typically involves several stages, including compiling the Gosu code, running unit tests (GUnit), and finally packaging the application into a Docker container. Manually triggering the build (Step E) involves clicking the" Run" button, often after specifying a specific branch or commit hash to ensure the resulting image contains the correct code changes.
This process is distinct from Lifecycle Manager (LCM), which is used for managing environment-specific configurations (like database settings or API keys) and promoting already-built images to specific" Planets" (Dev, Pre-Prod, etc.). Datadog (Option D) is used for post-deployment observability and would not be used to trigger an image build. Therefore, the combination of accessing the correct tool and manually initiating the build configuration is the verified procedure for cloud developers.

3.Succeed Insurance needs to add a new getter property to the Java class generated from the Contact entity.
According to best practices, what steps below would allow this to get implemented? (Select Two)
A. Add the enhancement definition to the Contact.eti file.
B. Add the enhancement definition to a new Contact.etx file.
C. Create a new Gosu enhancement for the Contact entity in the gw.entity.enhancements package.
D. Add a new get property to the enhancement.
E. Create a new Gosu enhancement for the Contact entity in the si.cc.entity.enhancements package.
F. Add a new get function to the enhancement.
Answer: D E
Explanation:
In Guidewire development, you cannot directly modify the underlying Java classes generated from entities. To add custom logic, properties, or methods to an existing entity like Contact, developers must use Gosu Enhancements. This allows the extra functionality to be available on every instance of that entity throughout the application (Rules, PCFs, and other Gosu classes) without altering the base product files.
1. Package Naming Standards (Option E)
According to the InsuranceSuite Developer Fundamentals and Cloud Delivery Standards, custom code must always be placed in a unique, customer-specific package. The gw package (Option C) is strictly reserved for Guidewire's internal code. Placing custom enhancements in a package like si.cc.entity.enhancements (where si stands for Succeed Insurance) ensures that the code is" upgrade-safe." During a platform upgrade, Guidewire replaces the gw packages but leaves the customer's custom packages untouched.
2. Properties vs. Functions (Option D)
The requirement specifically asks for a" getter property." In Gosu, this is implemented using the property get keyword. While you could technically write a function (e.g., getSomeValue()), a property allows for a cleaner syntax in other parts of the application. For example, if you define a property FullName_Ext, you can access it as myContact.FullName_Ext rather than myContact.getFullName_Ext(). This follows the Guidewire best practice of making the entity model feel like a cohesive, POJO-like structure.
Why other options are incorrect:
Options A and B: .eti (Entity Interface) and .etx (Entity Extension) files are metadata files used to define

the database schema (columns, foreign keys, etc.). They are not used to write Gosu logic or enhancement definitions.
Option F: While a" get function" is valid Gosu, the question specifically asks for a" getter property," which has a distinct syntax (property get) in the Guidewire framework.
By creating an enhancement in a customer-specific package and using the property syntax, the developer ensures the code is performant, readable, and follows the strict architectural guidelines required for Guidewire Cloud.

4.In the Extensions folder, there is a typelist file named BusinessType.ttx containing three typecodes: Insurer, Broker, and Agency. The business analysts have requested an additional typecode: Reinsurer. How should this typecode be added?
A. Create a reinsurer_Ext typecode in BusinessType.ttx
B. Create a reinsurer typecode in BusinessType.ttx
C. Create a .ttx extension file and add a reinsurer_Ext typecode to it
D. Create a Reinsurer_Ext typelist with a reinsurer typecode
Answer: B
Explanation:
When managing Typelists (Guidewire's version of enumerations) in a Cloud-compliant environment, there is a distinct difference between adding codes to an existing typelist and creating an entirely new one.
According to the InsuranceSuite Developer Fundamentals curriculum, if you are adding a new code to an existing typelist that is already an extension (indicated by the .ttx file extension), you simply add the new code to that existing .ttx file.
One of the most common points of confusion is the use of the _Ext suffix. While new entities and new typelists require the _Ext suffix to follow Cloud Delivery Standards, individual typecodes (the values within the list) generally do not require the suffix unless the specific project guidelines demand it for every single metadata element. In standard Guidewire practice, a typecode named reinsurer (Option B) is sufficient and cleaner.
Option C is incorrect because you should not create multiple .ttx files for the same typelist; you should consolidate extensions into the existing one.
Option D is incorrect because it suggests creating a whole new list rather than modifying the existing one.
Option A is less ideal because adding _Ext to individual codes within a list is not a mandatory platform requirement and can make logic like TC_REINSURER_EXT cumbersome to write in Gosu.

5.Which rule is written in the correct form for a rule which sets the claim segment and leaves the ruleset?
A. CONDITION: (claim: entity.Claim) return claim.Segment == nullACTION: (claim: entity.Claim, actions: gw.rules.Action) claim.Segment = TC_AUTO_LOW; actions.exit()
B. CONDITION: (claim: entity.Claim) if (claim.Segment == null) {claim.Segment = TC_AUTO_LOW} ACTION: actions.exit()
C. CONDITION: (claim: entity.Claim) return claim.Segment == nullACTION: (claim: entity.Claim, actions: gw.rules.Action) claim.Segment == TC_AUTO_LOW; actions.exitRuleset()
D. CONDITION: (claim: entity.Claim) return claim.Segment = TC_AUTO_LOWACTION: (claim: entity. Claim, actions: gw.rules.Action) actions.exit()

Answer: A

Explanation:

The Guidewire Rules Engine uses a declarative” Condition/Action” structure. For a rule to function correctly and follow best practices, the logic must be strictly separated between these two sections.

In Option A, the Condition is a pure Boolean expression (claim.Segment == null). In Gosu rules, the condition must return a value of true or false. If true, the engine proceeds to the Action block. The Action block in Option A correctly performs the assignment (claim.Segment = TC_AUTO_LOW) using the single equals sign, which is the assignment operator in Gosu. Crucially, it then calls actions.exit(). This is the standard method provided by the gw.rules.Action class to terminate the current ruleset execution for the object in scope.

Option C is incorrect because it uses the comparison operator (==) in the Action block instead of the assignment operator (=), meaning the segment would never actually be set. Additionally, exitRuleset() is not the standard syntax; the engine uses actions.exit().

Option D is incorrect because it attempts to perform an assignment within the Condition block, which violates the architectural purpose of the condition. Understanding this separation is vital for developers to ensure that rules are both performant and logically sound, preventing” infinite loops” or skipped logic within the claim or policy processing lifecycle.