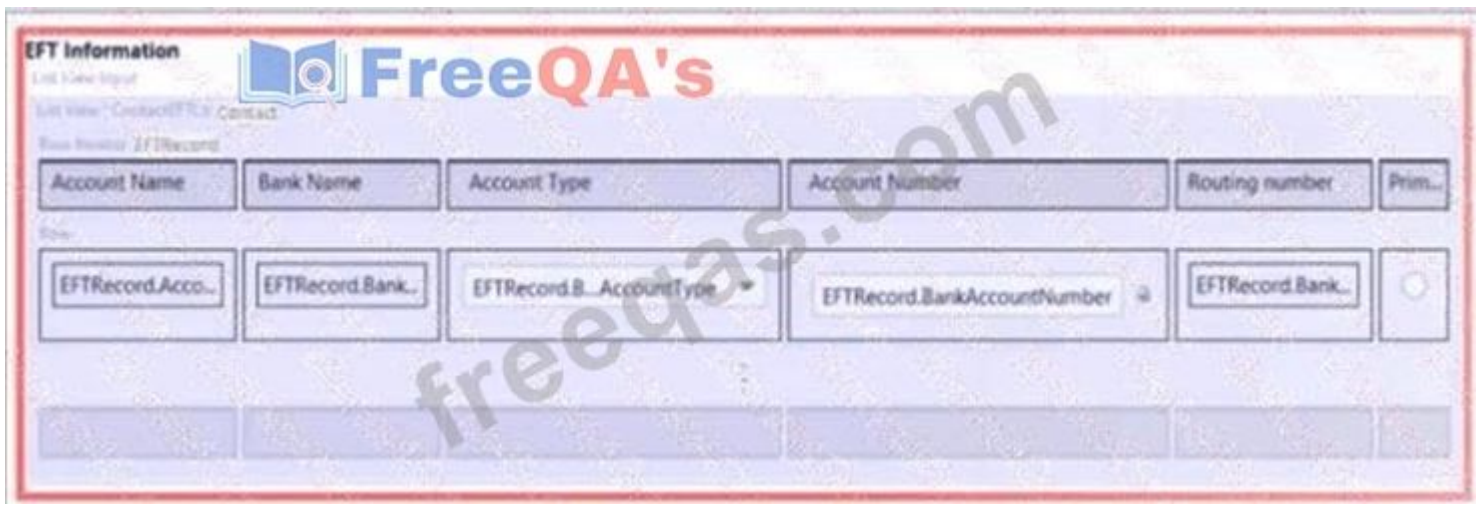
Guidewire.InsuranceSuite-Developer.v2026-08-03.q66

Exam Code:	InsuranceSuite-Developer
Exam Name:	Associate Certification - InsuranceSuite Developer - Mammoth Proctored Exam
Certification Provider:	Guidewire
Free Question Number:	66
Version:	v2026-08-03
# of views:	120
# of Questions views:	660
https://www.freeqas.com/qa/Guidewire/InsuranceSuite-Developer/Guidewire.InsuranceSuite-Developer.v2026-08-03.q66.html	

NEW QUESTION: 1
ContactManager provides an inline reference to an editable list view on the Contact Basics screen that supports adding and editing of banking information for contacts. The screenshot below shows this list view in Studio. There is an error within the red outline.



Which configuration changes are necessary to resolve the error? (Select two)
A. Add a toolbar widget to the list view input
B. Add and configure Iterator buttons
C. Replace the list view input with a panel ref
D. Replace the list view PCF with an inline list view
E. Add edit permissions to the row iterator
Answer: A,B (LEAVE A REPLY)
In the GuidewirePage Configuration Framework (PCF), displaying a list of data within aDetail View (DV) requires specific container widgets. When a developer uses aListViewInputto embed an existing List View into a form, they are essentially creating an "editable grid" section.
1. The Requirement for a Toolbar (Option A)
According to theInsuranceSuite Developer Fundamentalsguide, a ListViewInput is a specialized widget that acts as a wrapper for a List View. Unlike a standard List View displayed on its own page (which inherits the page's toolbar), a ListViewInput exists inside a Detail View column. For the list to be interactive- allowing users to add new bank accounts or remove existing ones-the ListViewInputmust have its own Toolbar. In Guidewire Studio, if

a ListViewInput is marked as editable but lacks a toolbar, the metadata validator will flag an error because there is no container to hold the necessary action buttons.
2. Configuring Iterator Buttons (Option B)
Once the Toolbar is added to the ListViewInput, it remains empty untilIterator Buttonsare placed inside it.
These buttons (typically the "Add" and "Remove" buttons) must be explicitly configured to point to theRow Iteratordefined within the referenced List View PCF.
The error in the screenshot is resolved by:
* Selecting the ListViewInput in the PCF tree.
* Adding aToolbarchild widget.
* AddingIterator Buttons(Add/Remove) to that toolbar.
* Linking those buttons to the correct iterator ID.
This combination provides the end-user with the UI controls needed to manipulate the banking information array. Options C and D represent alternative ways to structure the UI but do not address the specific configuration error of the ListViewInput widget. Option E relates to security and runtime behavior, not the structural metadata requirements of the PCF layout engine.

NEW QUESTION: 2
What is a purpose of logging in deployed systems that follows best practices?
A. Capturing the significant events for auditing
B. Troubleshooting payments by logging bank account numbers
C. Recording the details of a user ' s login credentials
D. Saving the runtime database queries and procedures
Answer: A (LEAVE A REPLY)
In the context of Guidewire InsuranceSuite, logging serves as a critical diagnostic and security tool. However, it must be implemented with a strict focus on Compliance and Performance. According to the Gosu Rules and Logging curriculum, the primary purpose of logging in a production (deployed) environment is to capture significant business and technical events that provide an audit trail for system behavior. This includes tracking successful transaction completions, identifying failure points in integration calls, and recording administrative actions.
A significant portion of the training emphasizes what not to log. Options B and C describe the logging of Personally Identifiable Information (PII) and sensitive credentials (bank account numbers and passwords).
Logging such data is a severe violation of Security Best Practices and regulatory standards like GDPR, HIPAA, and PCI-DSS. Guidewire Cloud standards mandate that logs must be scrubbed of any sensitive data to prevent data leaks. Furthermore, logging every single database query (Option D) is generally discouraged in production because it creates massive " log bloat " and incurs a heavy performance penalty due to disk I/O.
By following the best practice of logging significant events, developers ensure that support teams have enough information to troubleshoot functional issues (using tools like CloudWatch or Datadog) without compromising customer privacy or degrading system responsiveness. Effective logging strikes a balance between visibility and security, ensuring that the " story " of a transaction can be reconstructed without exposing the sensitive data within it.

NEW QUESTION: 3
Succeed Insurance needs to modify an existing PolicyCenter typelist called PreferredContactMethod with new options. Following best practices, which of the following options would a developer use?
A. Create a typelist extension called PreferredContactMethod.ttx and add the options there.
B. Create a typelist extension called PreferredContactMethod.Ext.tti and add the options there.

C. Create a typelist extension called PreferredContactMethod.Ext.ttx and add the options there.
D. Modify the existing PreferredContactMethod.tti file and add the options there.
Answer: (SHOW ANSWER)
Guidewire uses a specific file naming convention to separate base product definitions from customer extensions. ForTypelists(which are essentially enums stored in the database), the base definition is stored in a .
tti (Typelist Interface) file.
According toCloud Delivery Standards, you never modify the base .tti file (Option D). Instead, to add new codes to an existing typelist, you create aTypelist Extensionfile with the.ttxsuffix (Option A). The file name must exactly match the base typelist name. Options B and C are incorrect because the _Ext suffix is required fornewentities or typelists, but forextendingan existing Guidewire typelist, the .ttx suffix is the standard mechanism that ensures the new codes are merged correctly with the original ones during a platform upgrade.

NEW QUESTION: 4
What configuration item is needed to add ABContact.Notes to PendingContactChangeView following best practices?

A. Add a viewEntityType for Note to PendingContactChangeView.etx
B. Add a viewEntityType for Note to PendingContactChangeView.eti
C. Set the value on the input widget to PendingContactChangeView.ABContact.Note
D. Create a viewEntity that includes Note
Answer: (SHOW ANSWER)
In Guidewire InsuranceSuite, View Entities are specialized data model objects used primarily to provide flattened, high-performance data for ListViews (LVs). They function similarly to a database view, joining multiple related entities into a single virtual record to minimize the number of database queries required when rendering a page.
The screenshot provided shows PendingContactChangeView.eti. As indicated by the " This is a read-only file
" warning in the Studio interface, this is a Base Application File. According to the Data Model Architecture and InsuranceSuite Developer Fundamentals, developers must never modify base files directly. Direct modifications to .eti files are not upgrade-safe and would be overwritten during any future Guidewire platform update, leading to significant maintenance issues.
To extend the metadata of a base view entity, the best practice is to use an Extension File, which carries the .

etx suffix. By adding a viewEntityType for Note within a PendingContactChangeView.etx file, the developer instructs the system to merge this custom field into the existing base view entity at runtime. This allows the new field to be accessible in Gosu and PCFs as if it were part of the original definition, while keeping the custom code isolated for easy upgrades.
Option C is incorrect because bypassing the View Entity by traversing through ABContact.Note directly in a PCF widget defeats the performance purpose of the View Entity and can lead to " N+1 " query performance bottlenecks. Option D is incorrect because creating a separate view entity is unnecessary and redundant when the current view entity can be easily extended. Therefore, Option A is the only verified best practice for maintaining a scalable and upgradeable Guidewire configuration.

NEW QUESTION: 5
In the screenshot below

A developer has added a tab labeled Delinquencies to the tab bar of BillingCenter. This tab will contain several pages. The first page in the tab will display a summary of the currently-selected delinquency, the second page will show the associated policy, and the third page will show the associated account.
What PCF container will be used to configure this requirement?
A. A location ref
B. A location group
C. A location ref iterator
D. A location
Answer: (SHOW ANSWER)
In the Guidewire Page Configuration Framework (PCF), locations are organized into a hierarchical structure to manage navigation and user context. When a requirement involves grouping multiple related pages under a single entry point-such as a Tab in the Tab Bar or a sidebar menu-the correct container to use is a Location Group.
1. The Role of a Location Group
A Location Group is a " non-leaf " node in the PCF navigation tree. It does not display content itself; instead, it contains other locations (Pages, Popups, or even other Location Groups). In the context of the " Delinquencies " tab, the Location Group serves as the parent container that defines:
* The Tab entry in the top-level navigation.
* The list of child pages (Summary, Policy, Account).
* The navigation menu (usually appearing on the left side of the screen) that allows users to switch between these three pages.
2. Why Other Options are Incorrect
* Option A (Location Ref): This is a widget inside a container (like a Location Group or a Section) that simply points to another location. It is a " pointer, " not the organizational container itself.
* Option C (Location Ref Iterator): This is used to dynamically generate a set of links or locations based on an array of data (e.g., a tab for each Open Claim). It is not the standard way to define a static three- page tab structure.
* Option D (Location): This is a generic term that encompasses Pages, Popups, and Worksheets. A single " Location " (specifically a Page) can only display one set of data. It cannot manage the multiple-page navigation required by the " Delinquencies " tab.
According to the InsuranceSuite Developer Fundamentals guide, using a Location Group ensures that the user ' s context (like the selected Delinquency ID) is maintained as they click through the different sub-pages within that group. This provides a seamless UX where the application " remembers " which record is being inspected even as the view changes.

NEW QUESTION: 6
Which of the following represents logging best practices? Select Two
A. Mask personally identifiable information (PII) before including it in a log message.
B. Set the logging level to "info" in the production environment.
C. Set the logging level to "debug" in the production environment when diagnosing a production issue.
D. Log all information that is necessary to diagnose the transaction.
E. Log every transaction to ensure a complete audit trail.
Answer: A,D (LEAVE A REPLY)
Effective logging in Guidewire InsuranceSuite is a balance between providing enough information for troubleshooting and maintaining system security and performance. Two of the most critical best practices involveData PrivacyandDiagnostic Context.
First,Masking PII(Option A) is a non-negotiable requirement for modern insurance applications, especially those running on the Guidewire Cloud Platform. Personally Identifiable Information, such as social security numbers, credit card details, or even specific personal names and addresses, must never appear as clear text in the application logs. Logs are often aggregated into secondary systems (like Datadog or CloudWatch) and viewed by a wide range of support personnel. If PII is not masked or removed before logging, the company risks failing compliance audits (GDPR, HIPAA, etc.) and exposing sensitive data.
Second, developers should strive tolog all information necessary to diagnose the transaction(Option D).
This means providing context, such as a PublicID, a specific TransactionID, or the state of an object at the time of an error. Without this context, log entries like "Error processing claim" are useless for troubleshooting. The goal is to provide enough detail so that a developer can understand the failure path without needing to step through the code in a debugger.
Other options are incorrect because they represent poor operational or performance choices. Setting the level to "debug" in production (Option C) can lead to severe performance degradation due to high I/O. While "info" (Option B) is a common default, it is not a "best practice" in the same functional sense as security and diagnosis. Finally, "logging every transaction" (Option E) is not the purpose of application logs; audit trails should be handled via the system's built-inHistoryorEvent Messagingtables, not the text-based log files, to avoid overwhelming the storage and performance of the application.

NEW QUESTION: 7
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 (LEAVE A REPLY)
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.

NEW QUESTION: 8
As a developer for Succeed Insurance, you have been given a requirement to add the following options to a ContactManager typelist BusinessType that was provided with the product:
* Auto Repair Shop
* Home Inspector
* Collection Agency
Following best practices, which of the following options correctly adds these options to the existing typelist?
A. Adding the following options to the existing BusinessType.ttx file:Code: auto_repair_shop, Code: home_inspector, Code: collection_agency
B. Adding the following options to the BusinessType.tti file:Code: auto_repair_shop_Ext, Code: home_inspector_Ext, Code: collection_agency_Ext
C. Adding the following options to a new BusinessType_Ext.tti file:Code: auto_repair_shop, Code: home_inspector, Code: collection_agency
D. Adding the following options to a new BusinessType_Ext.ttx file:Code: auto_repair_shop, Code: home_inspector, Code: collection_agency
Answer: (SHOW ANSWER)
In Guidewire InsuranceSuite, typelists are defined using two types of metadata files: .tti (Typelist Internal) and .ttx (Typelist Extension). The .tti files contain the base out-of-the-box (OOTB) codes provided by Guidewire and should never be modified by a developer. Direct modifications to base files are not upgrade- safe and violate the core architectural principles of the platform.
To add custom codes to an existing typelist, the best practice is to use the extension file associated with that typelist, which carries the .ttx suffix. If the file BusinessType.ttx already exists in the configuration module, the developer simply adds the new typecode elements to it. If it does not exist, the developer creates it with the same name as the base typelist. At runtime, the Guidewire platform performs a " metadata merge, " combining the base codes from the .tti with the custom codes from the .ttx.
Option D is incorrect because the extension file should not have _Ext appended to the filename itself; it must match the name of the base typelist exactly to be recognized by the system. Option C is incorrect because .tti files are reserved for Guidewire ' s internal use. By following the pattern in Option A, the developer ensures that the new options (Auto Repair Shop, Home Inspector, and Collection Agency) are seamlessly integrated into the application while maintaining a clean, upgradeable configuration. This is a fundamental concept in Data Model Configuration and metadata management.

NEW QUESTION: 9
This code sample performs poorly due to the use of dot notation with multiple array expansions: var lineItems = Claim.Exposures*.Transactions*.LineItems. What is the recommended best practice to improve the performance of this code?
A. Rewrite the code with a nested for loop to retrieve the results
B. Break the code into multiple gosu queries to retrieve the results for each array
C. Write a query that fetches the addresses into a collection then uses where() to filter the results

D. Replace the dot notation syntax in the code with ArrayLoader syntax

Answer: (SHOW ANSWER)

In Guidewire InsuranceSuite, the expansion operator (*) is a powerful Gosu feature used to flatten arrays and access properties across a collection. However, as noted in the Advanced Gosu and System Health & Quality curriculum, using multiple expansions in a single statement-especially across deep entity hierarchies like Claim - > Exposures - > Transactions - > LineItems-is a significant performance anti-pattern.

When this dot-notation traversal is executed, the application performs " lazy loading. " For every exposure, it fetches all transactions, and for every transaction, it fetches all line items. This creates the N+1 query problem, where the number of database roundtrips grows exponentially with the data volume. Furthermore, all these entities are loaded into the application server's memory and added to the current Bundle. This leads to " Bundle Bloat, " which increases memory pressure, slows down garbage collection, and can significantly degrade the performance of the specific web request or batch job.

The recommended best practice to resolve this is to use the ArrayLoader syntax (Option D). The ArrayLoader API is specifically designed to perform " eager loading. " It allows the developer to specify related arrays that should be loaded in bulk using optimized SQL joins or batch fetches. By using ArrayLoader, the developer can retrieve the necessary nested data in a single or highly reduced number of database operations, ensuring that the data is ready in memory before the logic attempts to access it. This eliminates the overhead of repeated lazy-loading calls and is the standard architectural solution for improving the performance of deep entity graph traversals in Guidewire.

NEW QUESTION: 10

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: (SHOW ANSWER)

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 anewTypelist 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 atypekey. 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 extendingexistingbase 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.

NEW QUESTION: 11
Business analysts have provided a requirement to store contacts' usernames in the Click-Clack social media website in a single field on the Contact entity. Which solution follows best practices and fulfills the requirement?
A. Extend the Contact entity with a field named ClickClack_Ext of type addressline
B. Extend the Contact entity with a field named ClickClack of type blob
C. Extend the Contact entity with a field named ClickClack of type shorttext
D. Extend the Contact entity with a field named ClickClack_Ext of type shorttext
Answer: D (LEAVE A REPLY)
InGuidewire InsuranceSuite, extending the data model to accommodate custom business requirements must follow strict architectural standards to ensure the application remains upgradeable and compliant withCloud Delivery Standards.
1. The Importance of the Naming Suffix (The _Ext rule)
The primary rule in Guidewire configuration is that any customer-added element (entities, fields, or typelists) mustbe suffixed with_Ext. As specified in theInsuranceSuite Developer Fundamentalscourse, this suffix serves as a "namespace" that prevents naming collisions with future base-product updates provided by Guidewire. If you were to name a field simply ClickClack (as in Options B and C), and a future Guidewire update introduced a field with the exact same name, the application server would fail to start due to metadata conflict. Therefore, the field must be named ClickClack_Ext.
2. Selecting the Correct Data Type
For a social media username, the developer must choose the most efficient and semantically appropriate data type.
* shorttext (Option D):This is the standard type for strings up to 60 characters. It is the most appropriate for a username, as it is indexed efficiently by the database and provides enough space for almost any social media handle.
* addressline (Option A):While this is also a string type (typically 60 characters), it is semantically intended for physical street addresses. Using it for social media handles is poor practice as it makes the metadata confusing for other developers.
* blob (Option B):This is used for "Binary Large Objects," such as images or documents. Using a blob for a simple text username would cause massive performance issues during searches and consume unnecessary database storage.
By choosingOption D, the developer ensures that the field is clearly identified as a custom extension and uses the most performant data type for the specific information being stored. This follows the "KISS" (Keep It Simple, Stupid) principle and Guidewire's automated quality gates for Cloud deployments.

NEW QUESTION: 12
The results of a Guidewire Profiler analysis on a web page showed a large unaccounted-for time. The developer cannot identify which block of code is taking up so much time by examining the profiler output.
Which approach can help to account for the large time spent and improve reading of the profiler output?
A. Identify the PCF file name in the profiler output.
B. Print out the timestamp before and after the code blocks.
C. Surround the blocks of code with profiler tags.
D. Create a function to calculate processing time in the class.
Answer: C (LEAVE A REPLY)
The Guidewire Profiler is a sophisticated tool used to capture the execution time of various system operations, such as database queries, PCF rendering, and rule execution. However, when complex Gosu logic or large loops are executed, the profiler may show a " gap " in the timeline-often referred to as unaccounted-for time.
This occurs because the default instrumentation of the profiler only hooks into specific system events; it does not automatically track every individual line of custom Gosu code.

To gain visibility into these " dark " areas, the developer should use Custom Profiler Tags. By wrapping specific segments of logic or expensive method calls with these tags, the developer manually instructs the profiler to track that specific block ' s entry and exit times. When the profile is later analyzed in the Guidewire Management Console, the previously unaccounted-for time will now be categorized under the label provided in the tag. This method is vastly superior to manual timestamp printing (Option B) because it integrates directly into the graphical representation of the Profiler, allowing the developer to see how the code block interacts with database " bundles " and other concurrent processes. It provides a holistic view of the execution stack, making it the standard best practice for performance tuning and bottleneck identification in both InsuranceSuite Developer Fundamentals and System Health modules.

NEW QUESTION: 13
Which statement is true about the Project Release branch for an implementation using Git?
A. It stores the current production code and is updated whenever the production system is updated
B. It is used by the implementation team to develop code for a specific release
C. It is used by the implementation team to stabilize the code for a specific release
D. It contains product releases from Guidewire
Answer: (SHOW ANSWER)
In the Guidewire Cloud Platform (GWCP) development lifecycle, effective source control management is essential for maintaining a stable path to production. Guidewire recommends a specific branching strategy tailored for InsuranceSuite implementations using Git (typically hosted in Bitbucket).
The Project Release branch (often named release/*) serves a very specific purpose: stabilization. According to the " Developing with Guidewire Cloud " course, the standard workflow involves developers working on feature branches and merging them into a develop or integration branch. Once a set of features is deemed complete for a specific deployment cycle, a Release branch is created.
The primary goal of this branch is to isolate the release-ready code from the ongoing, potentially volatile development occurring in the main integration branch. On the Release branch, the team performs final GUnit testing, regression testing, and bug fixes specifically identified during the QA phase for that version. No new features should be introduced here. This isolation ensures that the " Candidate for Production " is stable and that any fixes applied are strictly for high-priority issues.
Option A refers to the master or main branch, which holds the current production state. Option B describes the function of feature or development branches. Option D is incorrect because product releases from Guidewire are provided as base code updates, which are typically merged into the customer ' s repository rather than existing as a " Project Release " branch. By focusing on stabilization, the Release branch minimizes the risk of introducing " noise " or untested features into the final production deployment.

NEW QUESTION: 14
An InsuranceSuite implementation project is preparing for deployment to the Guidewire Cloud Platform.
Which two Cloud Delivery Standards must be met before deployment? (Select two)
A. There are no instances of single statements with multiple expansion operators(*)
B. GUnit tests must be combined into suites and executed in Studio prior to each code deployment
C. The default system user su is configured as the second argument of the runWithNewBundle method
D. New entities and new columns added to existing entities use a customer suffix such as _Si
E. Log files contain no PII (Personally Identifiable Information) as clear text
Answer: D,E (LEAVE A REPLY)

Moving to theGuidewire Cloud Platform (GWCP)introduces a set of mandatory "Cloud Delivery Standards" designed to ensure that customer implementations are secure, upgradeable, and performant. Two of the most critical pillars in these standards areMetadata Naming ConventionsandData Privacy/Security.
First, naming conventions (Option D) are essential for maintaining the "Upgrade-Safe" nature of the cloud. In Guidewire Cloud, the base product is updated frequently. To prevent custom metadata (new entities or columns) from conflicting with future Guidewire base product releases, all extensions must use a unique customer suffix. While the standard example is _Ext, insurers often use their specific company initials, such as _Si for Succeed Insurance. This ensures that the custom data model remains distinct from the gw namespace.
Second, theObservabilityandSecuritystandards strictly forbid the logging ofPersonally Identifiable Information (PII)in plain text (Option E). In the cloud, logs are aggregated and viewed via tools like CloudWatch or Kibana. If sensitive data like Social Security Numbers, Credit Card numbers, or personal addresses are logged as clear text, it constitutes a major security risk and a violation of compliance standards like GDPR or SOC2. Guidewire's automatedQuality Gateswill often block a deployment if it detects potential PII leakage in the Gosu logging code.
Options A and B are general development practices but not the primary delivery standards that block deployment. Option C is actually a security risk; hardcoding the "su" (Super User) in bundle management is generally discouraged in favor of more granular permission handling.

NEW QUESTION: 15
Which scenario follows best practices for user interface field-level validation?
A. Proposed changes to user passwords contain only alphanumeric characters.
B. Store different social media profile addresses, regardless of the social media site involved.
C. The city and state populate automatically whenever a US user enters their ZIP code.
D. The interest rate field is 0 whenever a down payment is greater than €5000.
Answer: A (LEAVE A REPLY)
In Guidewire PCF Configuration, field-level validation is used to ensure that the data entered by a user conforms to specific constraints (format, length, or character types) before the page is even committed to the database. According to the InsuranceSuite Developer curriculum, the best practice is to implement simple, immediate constraints at the field level to provide the user with rapid feedback.
Scenario A is a classic example of field-level validation. Ensuring that a password contains only alphanumeric characters can be enforced directly on the input widget using Input Masking or a Validation Expression within the PCF. This prevents invalid data from entering the system at the earliest possible stage. It improves the user experience by identifying the error immediately, rather than waiting for a full database commit to trigger a server-side validation rule.
In contrast, Scenario C describes Reflection or Post On Change behavior, which is a UI automation/helper feature rather than a validation check. Scenario D represents complex business logic or a " validation rule " that involves cross-field dependencies (interest rate vs. down payment); while this can be done in a PCF, it is more typically handled in a Validation Guide or a dedicated Gosu validation class to ensure the rule is enforced regardless of which UI screen is used. By focusing field-level validation on structural requirements like alphanumeric constraints, developers maintain a clear separation between data integrity (formatting) and business logic (eligibility/rules).

NEW QUESTION: 16
A developer needs to prepare their local configuration changes for inclusion in the shared Bitbucket repository. According to the training, which actions, performed using Git-based commands in a GWCP context, are essential for this process? (Choose 3)
A. Pulling the latest changes from the remote repository and rebasing the developer ' s commit(s).
B. Committing the changes locally.
C. Deploying the application to a development planet.
D. Creating a new build schedule in TeamCity.

E. Pushing the finished branch to the remote Bitbucket repository.
F. Configuring pre-promotion quality gates.
Answer: (SHOW ANSWER)
In the context of Developing in the Cloud and using the Guidewire Cloud Platform (GWCP), managing source code follows standard Git-based workflows integrated with Atlassian Bitbucket. For a developer to successfully share their local configuration changes with the rest of the team, they must follow a sequence that ensures code integrity and avoids " merge hell. " The first essential step is Committing the changes locally (Option B). This records the developer ' s progress in their local repository ' s history. However, because other developers may have pushed changes to the shared repository in the meantime, the developer must synchronize their local environment. This is achieved by Pulling the latest changes from the remote repository and rebasing (Option A). Rebasing is preferred in the Guidewire curriculum because it creates a clean, linear project history by moving the developer ' s custom commits to the " tip " of the updated master or feature branch. Finally, once the local branch is current and all conflicts are resolved, the developer must Push the finished branch to the remote Bitbucket repository (Option E). Only after the code is in Bitbucket can it be picked up by TeamCity for automated building and testing.
Deploying to a planet (Option C) and creating build schedules (Option D) are downstream activities that occur after the code has been successfully merged into the shared repository. Similarly, Quality Gates (Option F) are pre-configured environment standards rather than a step in the Git commit-and-push workflow. Adhering to the commit-rebase-push cycle is the verified best practice for collaborative cloud development.

NEW QUESTION: 17
Which GUnit test method adheres to the Guidewire naming standards?
A. whenAssigningActivityTest
B. claimSegmentationTest
C. testBulkInvoiceBatchJob
D. testThatQuoteIsGenerated
Answer: C (LEAVE A REPLY)
Testing is a core pillar of the InsuranceSuite Developer curriculum, specifically through the use of GUnit, Guidewire ' s specialized testing framework based on JUnit. To maintain a clean and searchable test suite, Guidewire enforces specific naming conventions for test methods within a TestCase class.
According to the Gosu Coding Standards, test methods should always begin with the lowercase prefix test.
This prefix is used by various automated tools and IDEs (like Guidewire Studio) to identify executable test segments. Following the prefix, the name should be in camelCase and should clearly describe the functionality or method being validated.
Option C, testBulkInvoiceBatchJob, is the correct format. It starts with the required prefix and uses a clear, concise description of the functional area being tested. Option A and B fail because they do not begin with the test prefix, which would likely result in the GUnit runner skipping those methods entirely. Option D, while starting with test, follows a more verbose, sentence-like structure (testThatQuoteIsGenerated) which is common in some BDD (Behavior Driven Development) frameworks but is less standard in traditional Guidewire GUnit development compared to the direct functional

naming seen in Option C. Adhering to these standards ensures that tests are easily identifiable during the CI/CD process and that the results reported in TeamCity are logically organized for the development team.

NEW QUESTION: 18
Which statements about Gosu package structure and naming conventions follow Guidewire best practices for customer implementations? (Choose 2)
A. Use underscores in package names (e.g., my_package).
B. Place all custom classes in a single custom package.
C. Use generic package names like com.company.
D. Include the product code (e.g., pc, cc, bc) in the package structure.
E. Group classes solely by functional area (e.g., util, entity, batch).
F. Use the customer code as the top-level package segment.
Answer: D,F (LEAVE A REPLY)
In Guidewire InsuranceSuite, maintaining a clean and standardized Gosu Package Structure is vital for long- term maintainability, readability, and avoiding namespace collisions during upgrades. Guidewire recommends a hierarchical package structure that clearly identifies the origin and purpose of the code.
The first best practice (Option F) is to use a unique Customer Code (e.g., acme) as the top-level package segment. This differentiates custom logic from the out-of-the-box (OOTB) code, which always resides in the gw package. By starting with a customer-specific identifier, developers ensure that their classes will never conflict with future Guidewire platform updates.
The second best practice (Option D) is to include the Product Code (e.g., pc for PolicyCenter, cc for ClaimCenter) in the next level of the hierarchy. A typical package name would look like acme.pc.util or acme.
cc.integration. Including the product code is particularly important for insurers running multiple products, as it allows for the sharing of common logic in a " core " package while keeping product-specific logic isolated.
Regarding the other options: Option A is incorrect because package names should use dots as separators, not underscores, and follow lowerCamelCase. Option C is discouraged because generic names like com.company are too broad and do not align with Guidewire ' s product-centric architecture. Option E is insufficient because while functional grouping is necessary, it must sit beneath the customer and product tiers. Following the customer.product.functionalarea pattern ensures the codebase is organized, searchable, and fully compliant with Guidewire Cloud Standards.

NEW QUESTION: 19
During an implementation, which Git branch contains code across all releases including code under active development?
A. Mainline branch
B. Master branch
C. Product release branch
D. Production branch
Answer: (SHOW ANSWER)
In the context of Source Control Management (SCM) for Guidewire implementations, especially within the Guidewire Cloud Platform (GWCP), a robust branching strategy is essential for coordinating work across multiple teams and releases.
The Mainline branch (sometimes referred to as the develop branch in Gitflow, though Guidewire training specifically uses the term " Mainline ") serves as the primary integration point for all active development. It represents the " trunk " of the code tree where all completed features, bug fixes,

and configuration changes are merged after passing initial testing. Because it contains the cumulative progress of all developers and workstreams, it acts as the source of truth for the current state of the application.
While Release branches (Option C) are used to stabilize a specific version of the code for deployment to UAT or Production, and the Master/Production branch (Options B and D) represents the code currently live in production, the Mainline is unique because it holds the code destined for future releases as well.
Following the SurePath methodology, developers create " Feature " or " User Story " branches from the Mainline. Once a story is complete and verified with GUnit tests, it is merged back into the Mainline. This ensures that the build chain in TeamCity is always testing the most recent, integrated version of the configuration. Maintaining the integrity of the Mainline branch is critical for the " Continuous Delivery " model required by the Guidewire Cloud.

NEW QUESTION: 20
An insurer ran the DBCC checks against a copy of their PolicyCenter production database in a non-production environment to check for errors before promoting the code to production. Three errors with high counts were found in the category " Data update and reconciliation. " What are two best practices for resolving the errors?
(Select two)
A. Identify any bad data and write a SQL script to correct the data; run the script immediately.
B. Wait to see if error counts increase; if they increase by more than 10%, fix the errors.
C. Promote the code to production and run the DBCCs again to see if the same errors are found.
D. Analyze the errors to determine the root cause and correct the code responsible for the errors.
E. Search the Knowledge Base on the Guidewire Community for solutions to the problems found.
Answer: D,E (LEAVE A REPLY)
Database Consistency Checks (DBCC) are a critical part of the System Health and Quality framework. When these checks return errors, especially in categories like " Data update and reconciliation, " it indicates a mismatch between the database ' s physical state and the application's metadata expectations.
According to Guidewire best practices, the first essential step (Option D) is root cause analysis. A developer must determine why the data is inconsistent. For example, if a high count of errors appears after a code change, it likely means a Gosu rule, an integration, or a newly added required field is not being populated correctly. Fixing the symptom with a SQL script without addressing the code responsible will only lead to the errors recurring. Addressing the logic ensures that the " faucet " of bad data is turned off before attempting to
" mop up " the existing data.
The second best practice (Option E) is leveraging the Guidewire Community and Knowledge Base. Many DBCC errors, particularly those related to base application upgrades or standard entities, are well- documented. Guidewire provides specific remediation scripts or configuration adjustments for known issues.
Options A, B, and C are strictly prohibited under Guidewire SurePath standards. Running SQL scripts " immediately " without analysis can lead to further corruption. Waiting for error counts to increase (Option B) ignores potential system instability. Promoting to production (Option C) while knowing errors exist is a violation of deployment quality gates and can lead to production downtime. By combining technical analysis with community resources, developers ensure a stable transition to the production environment.

NEW QUESTION: 21
Which of the following are true about Guidewire Inspections?
A. Inspections must be triggered manually using the Analyze toolbar option.
B. There are no inspections provided with the out of the box version of the product.

C. Developers can create custom inspections profile in Studio to include any customer specific standards that are to be enforced.
D. Inspections are run at the command line by running the gwb inspect.
E. Inspections run automatically in the Gosu editor as a background task.
Answer: (SHOW ANSWER)
Guidewire Inspections are a form of static code analysis integrated directly into Guidewire Studio (which is built on the IntelliJ IDEA platform). These inspections are designed to help developers identify code smells, performance bottlenecks, and violations of Gosu Coding Standards in real-time.
The most important operational fact about inspections (Option E) is that they run automatically as a background task within the Gosu editor. As a developer types code, the IDE continuously analyzes the syntax and logic. If a violation is found-such as an unused variable, a potentially null reference, or an inefficient query-the editor provides immediate visual feedback through highlights (like yellow warnings or red errors) and " gutter " icons. This allows for " Shift-Left " quality management, where issues are corrected the moment they are created, rather than during a later build or code review phase.
While Option C is technically true (profiles can be customized), Option E is the primary characteristic of the tool's behavior as described in the System Health and Quality training. Option B is false because Guidewire provides hundreds of out-of-the-box inspections specifically tailored for InsuranceSuite (e.g., checking for PII in logs or inefficient bundle usage). Option D is incorrect as the primary mode of interaction is the IDE, not a command-line tool. By leveraging these automatic background inspections, developers maintain a high level of code quality and adhere to the SurePath methodology throughout the development lifecycle.

NEW QUESTION: 22
Given the following code example:
Code snippet
var query = gw.api.database.Query.make(Claim)
query.compare(Claim#ClaimNumber, Equals, "123-45-6798")
var claim = query.select().AtMostOneRow
According to best practices, which logic returns notes with the topic of denial and filters on the database?
A. var notesQuery = gw.api.database.Query.make(Note); var denialNotes = notesQuery.select().where(\elt -
> elt.Topic==NoteTopicType.TC_DENIAL)
B. var denialNotes = claim.Notes.where(\elt -> elt.Topic==NoteTopicType.TC_DENIAL)
C. var notesQuery = gw.api.database.Query.make(Note); notesQuery.compare(Note#Topic, Equals, NoteTopicType.TC_DENIAL);
notesQuery.compare(Note#Claim, Equals, claim); var denialNotes = notesQuery.select()
D. var notesQuery = gw.api.database.Query.make(Note); notesQuery.compare(Note#Topic, Equals, NoteTopicType.TC_DENIAL); var denialNotes =
notesQuery.select()
Answer: C (LEAVE A REPLY)
Efficiency in Guidewire performance relies heavily on the "Database-First" principle. To fulfill the requirement of filtering notes by bothClaimandTopicspecifically on the database, a new query must be constructed using theQuery API.
Option C is the only correct answer because it uses the .compare() method to apply two specific filters:
* Topic Filter:It filters for the specific typecode TC_DENIAL.
* Claim Filter:It links the query to the specific claim object found in the previous step.
By setting these parametersbeforecalling .select(), Guidewire generates a single SQL statement: SELECT * FROM cc_note WHERE topic = 'denial' AND claimid = ... The database performs the heavy lifting and returns only the relevant records.
Options A and B areanti-patterns. They fetch all notes (Option B) or execute a broad query (Option A) and then use the Gosu .where() method to filter in the application server's memory. This is highly inefficient.

Option D is incomplete as it would returneverydenial note in the entire system, regardless of which claim it belongs to.

NEW QUESTION: 23
A developer runs Database Consistency Checks for a new ClaimCenter release in a QA environment running a copy of the production database. Analysis of the output identifies data errors in both the QA and production data. Which two options follow best practices for correcting the data? (Select two)
A. Use the Production Data Fix Tool to correct production data
B. Write a Gosu query and run it in Scratchpad to correct the data
C. Export the data to a file, correct it, and run the Import Data Utility
D. Write a Gosu script and request that Guidewire Support review it
E. Contact the insurer's database group for a SQL script and test it in QA
Answer: D,E (LEAVE A REPLY)
Maintaining data integrity is the highest priority in a Guidewire implementation. WhenDatabase Consistency Checksidentify errors that exist in the production environment, the resolution process must be rigorous, audited, and safe.
According to theGuidewire System Health & Qualitycurriculum, developers should never attempt to fix production data errors using "ad-hoc" methods like Scratchpad (Option B) or standard Import utilities (Option C), as these bypass the complex referential integrity and validation logic inherent in the application.
Instead, the best practice involves two paths. First, for errors rooted in complex application logic, the developer shouldwrite a Gosu script and request a review from Guidewire Support(Option D). Guidewire Support provides a "Data Fix" service to ensure the script won't have unintended side effects on the database schema or application stability.
Second, if the error is purely at the database level (e.g., a broken foreign key or orphan record), the developer shouldwork with the insurer's database group to create a SQL script(Option E). However, this scriptmust be tested in a QA environment(that uses a copy of production data) before execution to verify that it correctly resolves the consistency check failure without damaging other data relationships. Option A is incorrect because "Production Data Fix Tool" is not a standard standalone tool name provided in the base product for this purpose.

NEW QUESTION: 24
An insurer has extended the ABContact entity in ContactManager with an array of Notes to capture information of interest about the contact over time. A developer has been asked to write a function to process all the notes for a given contact. Which code satisfies the requirement and follows best practices?
A. while (exists (note in anABContact.Notes)) { //do something }
B. for (note in anABContact.Notes) { //do something }
C. for (i = 1..anABContact.Notes.length) { //do something }
D. var aNote = anABContact.Notes.firstWhere(\ note - > note.Author != null) //do something
Answer: B (LEAVE A REPLY)
Gosu is a powerful, statically typed language designed to work seamlessly with the Guidewire data model.
When a developer needs to iterate over an Entity Array-such as the Notes array on an ABContact- following the most readable and efficient syntax is a core requirement of InsuranceSuite Developer Fundamentals.
The for..in loop (Option B) is the idiomatic " best practice " in Gosu for iterating through collections. This syntax is clean, prevents " off-by-one " errors common in index-based loops, and automatically handles the iterator logic behind the scenes. In this example, note serves as the element variable that represents a single Note entity in each iteration of the loop. This approach is highly optimized for the Guidewire Bundle and Query API, ensuring that the system can efficiently manage the memory objects as the loop progresses.

Other options represent suboptimal or incorrect patterns:
* Option A: Uses a while loop with an exists check, which is syntactically incorrect for iterating through an entire list and would likely lead to an infinite loop or a compilation error.
* Option C: Uses a range-based loop with an index. This is less readable and more prone to error, especially since arrays in Gosu are 0-indexed, but the range starts at 1.
* Option D: Uses the firstWhere enhancement, which only returns the first matching element rather than processing all notes as required by the business analyst.
By using the standard for loop, the developer ensures that the code is maintainable, follows Gosu Coding Standards, and is easily understood by other Guidewire developers.

NEW QUESTION: 25
A developer has finished a bug fix. Which step is needed before merging to follow best practices?
A. Recreate the development branch
B. Merge user story branch into parent branch
C. Clone the parent branch
D. Integrate parent branch to defect branch
Answer: D (LEAVE A REPLY)
In the Guidewire Cloud (GWCP) development lifecycle, managing code through Source Control Management (SCM) requires a disciplined branching strategy. When a developer completes a bug fix on a " defect " or " feature " branch, the environment is often dynamic, meaning other developers may have merged changes into the Parent Branch (such as develop or master) while the fix was being worked on.
The critical best practice before attempting to merge the fix back into the main codebase is to Integrate the parent branch into the defect branch (Option D). This is typically achieved through a git merge or a git rebase operation. The purpose of this step is twofold:
* Conflict Resolution: It allows the developer to identify and resolve any code conflicts locally on their branch where they have full context of their changes.
* Validation: It ensures that the bug fix is compatible with the most recent version of the application. By integrating the parent branch first, the developer can run GUnit tests and local builds against the combined code, ensuring that their merge will not " break the build " for the rest of the team.
Merging a branch that is " out of sync " with its parent directly into the main repository (Option B) often leads to failed builds in TeamCity and disrupts the CI/CD pipeline. Options A and C are administrative or redundant tasks that do not address the logical synchronization of the code. Adhering to this " pull-before- push " integration pattern is a cornerstone of the InsuranceSuite Developer curriculum for cloud-native delivery.

NEW QUESTION: 26
Given the following code sample:
gw.transaction.Transaction.runWithNewBundle(\newBundle - > {
var targetCo = gw.api.database.Query.make(ABCompany)
targetCo.compare(ABCompany#Name, Equals, " Acme Brick Co. ")
var company = targetCo.select().AtMostOneRow
company.Notes = " some value "
}, " su ")
What two items should be added or changed to follow best practices? (Select two)
A. company = newBundle.add(company) before setting company.Notes.

B. bundle.commit() after setting company.Notes.
C. Pass a value for the user (ensure the user is correctly specified).
D. Replace Equals with Includes in the comparison.
E. targetCo = newBundle.add(targetCo) after the compare statement.
Answer: A,C (LEAVE A REPLY)
This scenario highlights critical aspects of Bundle Management and transaction handling in Guidewire. The first and most significant issue is the modification of the company entity. In Guidewire, entities retrieved via a Query are typically " read-only " in their initial state. To modify an existing entity within a transaction, it must be explicitly associated with the current bundle. The instruction company = newBundle.add(company) clones the entity into the newBundle, making it editable. Without this step, attempting to set company.Notes would result in a runtime exception because the object is not " in the bundle. " Secondly, although the snippet shows " su " , the best practice for runWithNewBundle is to always ensure a valid, non-null user is passed to provide the necessary security context for the transaction. In many development scenarios, hardcoding " su " (Super User) is considered a placeholder, and production-ready code should dynamically resolve the appropriate user or ensure the execution context is valid.
Regarding the other options: Option B is incorrect because runWithNewBundle automatically handles the commit() operation at the end of the code block; manually calling it is redundant and can cause errors. Option E is a technical misunderstanding of the API, as the Query object (targetCo) is a tool used to find data and is never " added " to a database bundle. By following the pattern of adding the entity to the bundle and ensuring proper user context, the developer adheres to the core principles of Gosu Bundle Management and data integrity.

NEW QUESTION: 27
Which logging statement follows best practice?
A. logger.error(DisplayKey.get(" Web.ContactManager.Error.GeneralException " , e.Message))
B. logger.info(logPrefix + " [Address#AddressLine1 = " + address.AddressLine1 + "] [Address#City " + address.City + "] [Address#State " + address.State + "] ")
C. if(logger.InfoEnabled) { logger.debug(" Adding " + contact.PublicID + " to ContactManager ") }
D. if(logger.DebugEnabled) { logger.debug(logPrefix + someReallyExpensiveOperation()) }
Answer: D (LEAVE A REPLY)
In Guidewire InsuranceSuite, logging is a critical tool for production support, but it must be implemented with strict attention to performance and data privacy. Option D represents the gold standard for performance- conscious logging in Gosu. When a developer needs to log a message that involves a " really expensive operation " (such as a complex string concatenation, a database lookup, or a heavy calculation), they should always wrap the logging call in an if statement that checks if that specific log level is enabled. Without this check, the Gosu engine would execute someReallyExpensiveOperation() to construct the string argument even if the logging level is set to " Info " and the " Debug " message is ultimately discarded. This can lead to significant, unnecessary CPU overhead in production environments.
Furthermore, other options violate key architectural principles. Option B is a significant security risk as it logs Personally Identifiable Information (PII) like address lines and cities; Guidewire Cloud standards strictly forbid logging PII to ensure compliance with privacy regulations like GDPR and CCPA. Option C contains a logical mismatch where the developer checks for InfoEnabled but attempts to log at a debug level. Option A is suboptimal because it passes e.Message as a string rather than passing the exception object itself, which prevents the logger from capturing the full stack trace. By following the pattern in Option D, developers ensure the application remains performant while providing necessary diagnostic data only when explicitly requested through configuration.

NEW QUESTION: 28
Which two performance issues of Guidewire core products can be analyzed using the Guidewire Profiler?
(Select two)

A. User Interface
B. Batch process
C. Database index
D. Data warehouse
E. Client reflection
Answer: A,B (LEAVE A REPLY)
The Guidewire Profiler is a diagnostic tool designed to help developers identify and resolve performance bottlenecks within the application. It captures detailed " stacks " of execution, showing exactly how much time is spent in Gosu logic, PCF rendering, and database queries.
According to the System Health & Quality training, the Profiler is primarily triggered through specific entry points. The User Interface (Option A) is the most common entry point; by enabling the profiler for a specific web session, a developer can analyze the performance of individual PCF pages, identifying slow-loading widgets or inefficient page queries. Batch Processes (Option B) are the second critical entry point. Since batch jobs (like the Billing/Claims renewal cycles or escalation tasks) often process massive amounts of data, profiling them allows developers to see where the job is lagging-whether it is in the " find " logic or the " execution " logic.
Options C and D are incorrect because the Profiler does not directly analyze the Database Index or the Data Warehouse as standalone entities. While the Profiler can show that a query is slow, determining why (such as a missing index) requires external database management tools or Guidewire's Database Health tools.
Similarly, Client Reflection (Option E) is a UI behavior that occurs in the browser, whereas the Profiler tracks server-side execution. Mastering the use of the Profiler for UI and Batch processes is a fundamental skill for maintaining high system quality in production.

NEW QUESTION: 29
There is a requirement for an additional filter on Desktop Activities. The filter requires a complex query.
Which option will meet the requirement and follow best practices?
A. Use a Gosu query with a " where " clause to find the needed rows, then pass the results to the filter.
B. Use a " for " loop with an " if " statement to find the needed rows, then pass the results to the filter.
C. Create a function using a Gosu query and a subselect clause, then call that function from the filter property.
D. Create a function using a Gosu query and compare clauses, then call that function from the filter property.
Answer: (SHOW ANSWER)
In Guidewire InsuranceSuite, implementing filters on high-volume pages like Desktop Activities requires a deep understanding of performance and the Gosu Query API. When a business requirement involves a " complex query " -typically meaning logic that spans across multiple related entities or involves conditional logic that cannot be expressed in a simple ToolbarFilterOption-the best practice is to encapsulate that logic within a separate Gosu function.
Using a function that utilizes a subselect clause is a preferred architectural pattern for complex filtering. A subselect allows the database engine to perform the filtering logic entirely at the SQL level (e.g., using an EXISTS or IN clause) without pulling large amounts of data into the application server ' s memory. This is significantly more efficient than retrieving a list of objects and then filtering them in Gosu (as suggested in Option A or B).
In the context of PCF Configuration, the filter property of a ToolbarFilterOption or a Filter widget can call this function, which returns a Query object. By returning a Query instead of a list of results, the UI component can further optimize the database call with paging and sorting logic. Option B, which suggests using for loops and if statements, is a direct violation of performance standards as it would result in " N+1 " query problems and excessive memory consumption on the application tier. Therefore, leveraging the power of the Query API with subselects ensures that the application remains responsive even as the volume of activities grows.

NEW QUESTION: 30

The following screenshot displays a segment of the menu items in the sidebar on a Guidewire application:

[Financials, Notes, Documents, Plan of Action, Services, Litigation, History]. The business analysts have uncovered a requirement that the Documents, History, Litigation, and Notes pages should be grouped under a single heading, to be called Legal Records. What is the best practice for accomplishing this?

A. Place the views associated with those four pages onto a single screen named Legal Records, and place that screen on the sidebar, removing the individual pages from the sidebar

B. Create a location group named Legal Records, containing those four items, and add it to the sidebar, removing the individual pages from the sidebar

C. Click the gear icon in the UI, select user settings, select Sidebar, set preferences for the four pages according to the desired view

D. Rearrange the order of the sidebar so that the four pages are all together, set the indent property for all four pages, and insert a label above them, which reads Legal Records

Answer: B (LEAVE A REPLY)

In the Guidewire InsuranceSuite UI architecture, the organization of navigation elements is governed by the hierarchy of PCF Locations. A location represents a destination in the application, such as a Page, a Worksheet, or a Wizard. When requirements call for grouping multiple related pages into a logical, hierarchical structure within the sidebar or a tab set, the standard architectural component used is the Location Group.

A Location Group acts as a container for other locations (Pages, Forwards, or even nested Location Groups).

According to PCF Architecture best practices, creating a Location Group named " Legal Records " and nesting the Documents, History, Litigation, and Notes pages within it allows the UI engine to render them as sub-items under a single, expandable heading. This maintains a clean and organized sidebar, which is critical for usability as the complexity of an application grows. Once the Location Group is defined and added to the Sidebar PCF, the individual page references are removed from the top-level sidebar configuration to prevent redundancy.

In contrast, Option A is incorrect because a " Screen " is a container for UI widgets (like DetailViews and ListViews) within a page, not a navigation location itself. Option C refers to user preferences, which do not change the underlying application configuration or satisfy structural business requirements. Option D describes a manual styling approach that does not leverage the built-in navigation framework; using indents and labels manually is not upgrade-safe and fails to provide the true modal navigation behavior (such as showing the group as " active " when any child page is selected) provided by a proper Location Group.

Therefore, using a Location Group is the only verified best practice for grouping sidebar navigation items.

NEW QUESTION: 31

A developer is creating an entity for home inspections that contains a field for the inspection date. Which configuration of the file name and the field name fulfills the requirement and follows best practices?

A. HomeInspection.etx, InspectionDate.Ext
B. HomeInspection_Ext.eti, InspectionDate.Ext
C. HomeInspection_Ext.etx, InspectionDate
D. HomeInspection.eti, InspectionDate.Ext
E. HomeInspection.Ext.eti, InspectionDate
Answer: B (LEAVE A REPLY)
Guidewire'sMetadata Naming Conventionsare strictly enforced to ensure that customer code remains distinct from Guidewire's base product code, which is essential for seamless platform upgrades.
When creating abrand-new entity, the developer must use the.eti (Entity Interface)extension. Following Cloud Delivery Standards, the entity name itself must include the_Extsuffix. Therefore, HomeInspection_Ext.
eti is the correct file structure. Regarding the fieldswithinthat custom entity, Guidewire best practices recommend applying the_Extsuffix to custom columns as well (Option B), even if the entity itself is custom.
This provides a consistent visual indicator in Gosu code that the developer is interacting with an extension rather than a base product element. Option A and C use the .etx extension, which is reserved forextending existing base entities(e.g., adding a field to Claim). Option D is incorrect because it lacks the mandatory suffix on the entity name. Option E uses an invalid file naming format. Following the convention in Option B ensures the data model is compliant with Guidewire's automated quality gates.

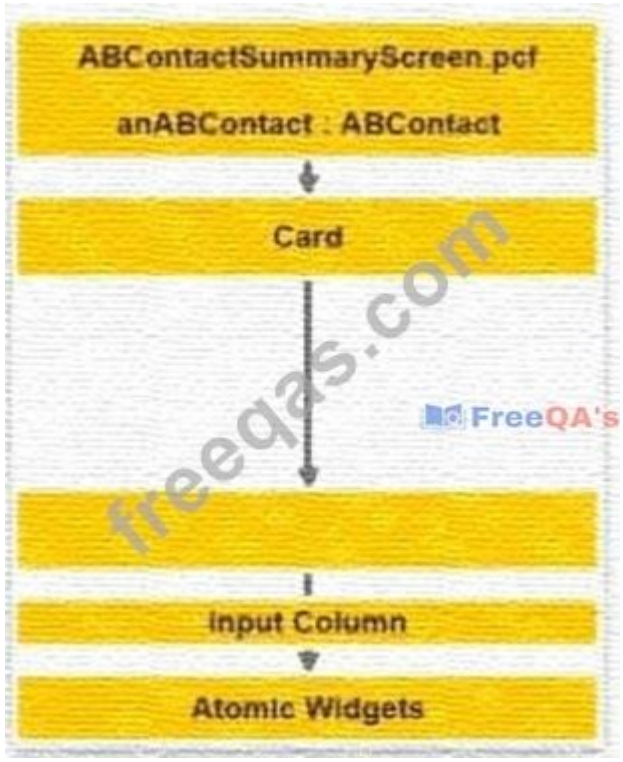
NEW QUESTION: 32
An insurer has a new requirement for capturing information for doctors added to ContactManager. In the Additional Info section of the Basics tab, a new Contact Picker for Hospital Affiliation will be added. When updated, the hospital address should update automatically.

What is the appropriate configuration approach to satisfy these requirements?
A. Enable Post On Change for the Hospital Affiliation field so changes are broadcast to the server and the browser redraws fields requiring changes

B. Add an expression to the visible property of the hospital address that evaluates to true when the value of the Hospital Affiliation field changes
C. Write a validation expression for the hospital address field that prevents the record from updating if it does not match the selected Hospital Affiliation
D. Create a Preupdate rule that updates the hospital address when changes to the Hospital Affiliation field are committed
Answer: A (LEAVE A REPLY)
In Guidewire InsuranceSuite, the concept of Dynamic UI is used to create a responsive user experience where the state of the interface changes based on user input. When a requirement specifically asks for a field to update " automatically " in the user interface based on a selection made in another field, the standard best practice is to leverage the Post On Change property.
By default, data entered into a widget is only sent to the server when a " main " action occurs, such as clicking an " Update " or " Next " button. However, when Post On Change is enabled for a specific widget (in this case, the Hospital Affiliation picker), any change to its value triggers an immediate server-side round trip.
This broadcast allows the server to execute any associated Gosu logic or " Reflection " properties. The server then recalculates the values of dependent fields-such as the Hospital Address-and instructs the browser to redraw only the affected portions of the page. This ensures that the user sees the updated address immediately without needing to manually refresh or save the record.
Other options, while valid for different contexts, do not satisfy the " automatic update " requirement.
Visibility logic (Option B) only controls whether a field is shown, not its content. Validation expressions (Option C) are used to block invalid data from being saved, which is a defensive measure rather than an automation feature. Finally, Preupdate rules (Option D) execute only when the data is being committed to the database at the end of a transaction. While a Preupdate rule could theoretically sync the data, the user would not see the update reflected in the UI while they are still editing the page. Therefore, following the PCF Configuration lessons on Dynamic UI, enabling Post On Change is the correct architectural approach.

NEW QUESTION: 33
Succeed Insurance has a page in PolicyCenter with a large fleet of vehicles. They want multiple filters to show only a subset of vehicles. Which methods follow best practices?
A. Apply the filter using the Row Iterator configuration in the PCF.
B. Implement filtering logic in the list view PCF using visible properties.
C. Add multiple Filter Options using Gosu Standard Query Filters.
D. Add a ListView Filter widget to the ListView.
E. Retrieve all policies and filter them in the application server layer.
F. Use Gosu's where method on the retrieved collection in memory.
Answer: C (LEAVE A REPLY)
When dealing with alarge fleet of vehicles, performance is the primary concern. Retrieving thousands of vehicle records and filtering them in the application server's memory (Options E and F) is a high-risk anti- pattern that leads to latency and high memory consumption.
The best practice for implementing efficient UI filters on large datasets is to useGosu Standard Query Filters (Option C). These filters are added to the ListView's toolbar. When a user selects a filter (e.g., "Only Heavy Trucks"), the Guidewire platform translates that filter into a SQL WHERE clause. This allows thedatabaseto do the work, returning only the specific subset of vehicles requested. This "Database-First" approach ensures that the application server remains responsive and that the network traffic between the database and the application is kept to a minimum.
Option A (filtering on the Row Iterator) and Option B (using "visible" properties) still require the system to fetch all the data from the database first, which does not solve the underlying performance issue. Using Query Filters is the only scalable solution for InsuranceSuite applications managing high-volume data.

NEW QUESTION: 34
Given the image:



Which container type must be added between Card and Input Column?
A. Detail View PCF File
B. Detail View
C. List View
D. Input Set
Answer: B (LEAVE A REPLY)
The Guidewire Page Configuration Framework (PCF) follows a strict nesting hierarchy to ensure that the layout engine can correctly render widgets on the screen. According to the InsuranceSuite Developer Fundamentals curriculum, specifically the lesson on " Container Widget Usage, " developers must understand the parent-child relationships required for different layout styles.
A Card widget is a component of a CardViewPanel, used to create tabbed interfaces within a page. However, a Card itself cannot directly host an Input Column. Instead, a Card serves as a container for other panels. To display data fields in the standard column-based layout favored by InsuranceSuite, a DetailViewPanel (commonly referred to simply as a Detail View in the Studio palette) must be placed inside the Card.
The Detail View acts as the intermediate container that establishes the data context (the row or entity being edited) and provides the grid system necessary for the Input Column. The Input Column, in turn, allows developers to align fields vertically. Without the Detail View container, the PCF would be syntactically invalid because the layout engine requires the Detail View to manage the labels and input alignment for any child columns.
Option A is incorrect because a " PCF File " is the entire document, not a widget added to a tree. Option C (List View) is used for tabular data, not column-based input layouts. Option D (Input Set) is a grouping mechanism that sits inside or alongside an Input Column but cannot serve as the parent to one. Therefore, adding a Detail View (B) is the correct and necessary step to bridge the hierarchy between the Card and its Input Columns.

NEW QUESTION: 35
An insurer wants to add a new typecode for a loan account to a base typelist, BankAccountType, that has not been extended. Which step must a developer take to perform this task following best practices?
A. Create a BankAccountType.ttx file and add a new typecode LoanAccount_Ext.

B. Create a BankAccountType.ttx file and add a new typecode LoanAccount.
C. Open the BankAccountType.tti and add a new typecode LoanAccount.
D. Create a BankAccountType_Ext.tti file and add a new typecode LoanAccount.
Answer: A (LEAVE A REPLY)
The Guidewire Data Model uses a strict separation between base application code and customer extensions.
Base typelists are defined in .tti (Typelist Internal) files, which are strictly off-limits for modification by developers (ruling out Option C).
To extend a base typelist, the developer must use a Typelist Extension file, which always uses the .ttx suffix.
The name of the .ttx file must match the name of the base typelist exactly (e.g., BankAccountType.ttx).
Adding _Ext to the filename (Option D) is incorrect, as the system would not recognize it as an extension of the base list.
Regarding the code itself, Guidewire best practices for InsuranceSuite Developer projects recommend adding a customer-specific suffix, such as _Ext, to any new typecodes added to a base typelist. This ensures that if a future Guidewire update introduces a " LoanAccount " code to the base product, the customer ' s custom code (LoanAccount_Ext) will not collide with it. Collisions can cause significant issues during upgrades, including database constraint violations and broken logic. By following the pattern in Option A, the developer ensures that the configuration is upgrade-safe, follows the SurePath methodology, and correctly integrates with the application ' s Open Type System.

NEW QUESTION: 36
An insurance carrier plans to launch a new product for various types of Recreational Vehicles (RVs)-such as motorhomes, boats, motorcycles, and jet skis. When collecting information to quote a policy, all RVs share some common details (like purchase date, price, year, make, and model), but each type also has its own unique properties. According to best practices, what should be done to configure the User Interface so that only the relevant RV details are shown when creating a policy quote? Select Two
A. Create a separate page for each type of RV.
B. Create a Modal Input Set for each RV type.
C. Create separate inline Input Sets for each RV type and set the visibility on each Input Set
D. Create a Detail View that includes the properties that are common to all of the RV types.
E. Place an Input Set Ref on the Detail View and configure the RV type as the Mode.
F. Define a Location Group to allow the user to choose the page for each RV type.
Answer: (SHOW ANSWER)
In the Guidewire Page Configuration Framework (PCF), the primary goal for handling polymorphic data- such as a base Recreational Vehicle entity with various subtypes-is to maximize code reuse while providing a dynamic user experience. According to the InsuranceSuite Developer Fundamentals course, the best practice for this scenario involves a " Master-Detail " design pattern utilizing Modal PCFs.
The first step (Option D) is to create a primary Detail View (DV). This DV acts as the foundation for the UI and contains all the fields that are shared across all RV types, such as PurchaseDate, Price, and Model. By centralizing these common fields, the developer ensures that any global changes to RV data (like adding a " Condition " field) only need to be made in one place, rather than across multiple fragmented pages.
The second step (Option E) addresses the unique properties of each RV type. Rather than cluttering the main DV with every possible field and using complex " visible " expressions (which is what Option C suggests and is discouraged due to performance and maintenance overhead), developers should use an Input Set Ref with the Mode property set. Each specific RV type (e.g., Boat, Motorcycle) has its own separate Input Set. At runtime, the Guidewire application looks at the RV type of the current object and automatically renders the corresponding Input Set. This " Modal " approach is the standard architectural way to handle subtypes in PolicyCenter and ClaimCenter. Options A, B, and F are incorrect because they either introduce unnecessary navigation complexity or fail to leverage the built-in dynamic rendering capabilities of the PCF framework.

NEW QUESTION: 37

An insurer stores the date a company was established in the company records. A business analyst identified a new requirement to calculate a company ' s years in business at the time a loss occurred. The years in business will be determined using the date established field and the claim date of loss.
The image below shows the Contact structure in the data model:

Which configuration steps will satisfy the requirement? (Select two)
A. Create a new enhancement class for the Company entity under the insurer package
B. Create a function to calculate the years In business in a Company enhancement
C. Create a setter property to calculate the years in business in the Contact enhancement
D. Create a new enhancement class for the Contact entity under the gw package
E. Create a function to calculate the years in business in a UI Helper class under the gw package
F. Create a getter property to calculate the years in business in a Company enhancement
Answer: A (LEAVE A REPLY)
In Guidewire development, the preferred way to extend base entities with business logic or derived data is through Gosu Enhancements. This approach allows you to add properties or methods to an entity that appear as if they were part of the original class.
1. Enhancement Location and Package (Option A)
According to the Guidewire InsuranceSuite Developer Fundamentals guide, any custom enhancement must be placed in a customer-specific package (e.g., si.pc.contact for Succeed Insurance). Using the gw package (Options D and E) is strictly prohibited as it is reserved for Guidewire ' s internal product code. Because " Date Established " is specific to the Company entity (as indicated in the Contact hierarchy), the enhancement should target the Company entity directly.
2. Using a Getter Property (Option G)
The requirement is to " calculate " a value based on existing data. The most efficient and readable way to implement this in Gosu is via a getter property (property get). Unlike a standard function (Option B), a getter property allows you to access the value in PCFs or rules using simple dot notation (e.g., myCompany.
YearsInBusiness_Ext), making the code cleaner and more maintainable.
Why other options are incorrect:
* Option B: While a function would technically work, a getter property is the best practice for a value that logically represents a " read-only " attribute of the entity.
* Option C: A setter is used to write data to a field. Since " Years in Business " is a derived calculation, it should not be manually set; it should be calculated on-the-fly from the source date fields.
* Options D and E: As mentioned, these use the gw package, which violates upgrade-safety standards and would cause the " Cloud Assurance " checks to fail.
By creating a Company enhancement in the customer ' s package and providing a property get, the developer creates a reusable, performant solution that follows the platform ' s core architectural principles.

NEW QUESTION: 38

The following Gosu statement is the Action part of a validation rule:

```
claim.rejectField("State", TC_PAYMENT,
  DisplayKey.get("Rules.Validation.Claim.NotInDraft",
  null, null)
```

It produces the following compilation error:
Gosu compiler: Wrong number of arguments to function rejectFieldQava.lang.String, typekey.
ValidationLevel, java.lang.string, typekey.ValidationLevel, java.lang.string). Expected 5, got 3 What needs to be added to or deleted from the statement to clear the error?

A. The two nulls must be replaced with a typekey and a string

B. A left parenthesis must be delete

C. The word "State' must be replaced with a DisplayKey

D. A right parenthesis must be added.

Answer: (SHOW ANSWER)

In GuidewireValidation Rules, the rejectField method is a critical tool for identifying specific fields that fail business logic checks. This method allows the application to highlight the exact UI widget in red and provide a specific error message to the user.
As indicated by the compiler error, the rejectField method on a Guidewire entity (like Contact or Claim) has a very specific signature that requiresfive parameters:
* Field Name (String):The name of the property being validated (e.g., "State").
* Validation Level (ValidationLevel):The severity of the failure (e.g., TC_LOADSAVE).
* Error Message (String):The text displayed to the user.
* Error Group (ValidationLevel):An optional group for categorizing the error.
* Error ID (String):An optional unique identifier for the specific error.
When the compiler reports"Expected 5, got 3", it means the developer only provided the first three arguments. To resolve this error according to Guidewire best practices, the developer must complete the signature. While null is often passed for the final two arguments if they are not needed, the compiler requires them to be present so it can identify which version of the overloaded rejectField method is being called.
The reason Option A is the recognized answer in this context is that simply adding null, null is often insufficient if the types aren't explicitly recognized or if the code had "placeholder" nulls that didn't match the expected typekey/string types. By ensuring the 4th argument is aValidationLeveltypekey and the 5th is a String, the developer satisfies the Gosu compiler's strict type-checking requirements. This ensures the validation logic is correctly registered within the current bundle transaction and will properly interrupt the commit process if the condition is met.

NEW QUESTION: 39

A developer has designed a detail view with an email address input. What is the best practice for ensuring that only a properly formatted email address can be entered?

A. Create an email address class with a validation method

B. Use database validation for the email address

C. Usefield-level validationfor the email address

D. Create a validation rule for email addresses

Answer: (SHOW ANSWER)

For standard formatting requirements like phone numbers, ZIP codes, or email addresses, Guidewire recommendsField-Level Validation(Option C). This is implemented using the validationExpression property on the PCF widget or, more ideally, by associating aValidatorin the Data Model (.eti/.etx).

Field-level validation provides the best user experience because it triggers immediately when the user navigates away from the field (client-side or AJAX refresh), providing instant feedback. Using aValidation Rule(D) is a "heavier" server-side operation that only triggers when the user tries to save the entire page. By using a Regex-based validator at the field level, the application maintains data integrity with minimal performance overhead.

NEW QUESTION: 40
An analyst is examining the process for promoting a verified build from the development environment to production. Which statements accurately describe key steps in the flow of code changes between physical star systems in GWCP, according to the training? (Choose 2)
A. Production database backups are not required when promoting builds.
B. A build must be deployed to every planet in the dev star system before promotion is allowed.
C. Builds are promoted sequentially from dev to pre-production and then to production physical star systems.
D. Code is directly committed from dev to production repositories.
E. The Build Promotion app in Guidewire Home is used to manage the promotion process.
Answer: C,E (LEAVE A REPLY)
The Guidewire Cloud Platform (GWCP) enforces a rigorous and standardized path for code promotion to ensure maximum stability in production environments. This process follows the Astronomy Metaphor where code moves between logical partitions.
Statement C is a fundamental truth of the cloud delivery model: Sequential Promotion. Code cannot " skip " environments. A build must first be verified in the Development Star System (on a Dev planet). Once verified, that same immutable build (Docker image) is promoted to the Pre-Production Star System for User Acceptance Testing (UAT) and Performance testing. Only after passing the " Quality Gates " in Pre-Prod can the build be promoted to the Production Star System. This sequence ensures that the exact same code being deployed to production has been thoroughly vetted in lower environments.
Statement E identifies the tool used for this management: the Build Promotion app located in Guidewire Home. This application provides a centralized interface for authorized users to select a verified build from a lower " Orbit " and promote it to a higher one. This tool abstracts the underlying complexity of the CI/CD pipeline and ensures that the promotion follows all security and compliance protocols.
Option D is incorrect because code is never " committed " directly to production; rather, a pre-compiled build image is promoted. Option B is incorrect as not every dev planet needs a deployment-only the specific verified " golden " build needs to move forward. Adhering to this sequential, tool-managed process is a key requirement of the Guidewire Cloud Standards.

NEW QUESTION: 41
A developer has completed a configuration change in an InsuranceSuite application on their local environment. According to the development lifecycle described in the training, which initial steps are required to move this change towards testing and deployment? Select Two
A. Deploy the application directly to a pre-production planet.
B. Schedule automated builds in TeamCity
C. Push the code changes to the remote source code repository in Bitbucket.
D. Trigger a TeamCity build via Guidewire Home if it has not already begun automatically.
E. Create a new physical star system in Guidewire Home.
F. Configure pre-merge quality gates in Bitbucket.
Answer: (SHOW ANSWER)
The Guidewire Cloud Platform (GWCP) development lifecycle is built around a modern CI/CD (Continuous Integration/Continuous Delivery) pipeline. This process moves code from a developer ' s local workstation through various " Planets " (environments) using integrated tools like Bitbucket, TeamCity, and Guidewire Home.

The first step in moving a local change toward production is committing and pushing the code to Bitbucket (Option C). Bitbucket serves as the centralized Git-based source code repository. This action triggers the " Build " phase of the lifecycle. Once the code is in Bitbucket, the next step involves the CI server, TeamCity.
TeamCity is responsible for compiling the Gosu code, running automated GUnit tests, and performing static code analysis (Quality Gates). While TeamCity is often configured to trigger automatically upon a push, a developer may need to manually trigger or monitor the build via Guidewire Home (Option D) if they need immediate feedback or if the automation is set to a specific schedule.
Options such as " Deploying directly to pre-production " (Option A) are impossible in the GWCP model, as code must first pass through the " Dev " planet and satisfy quality gates before being promoted. " Scheduling automated builds " (Option B) is an administrative task, not an initial step for a developer ' s specific change.
Finally, " creating a star system " (Option E) refers to the infrastructure setup usually handled by Guidewire Cloud operations, not a part of the standard code-change lifecycle. Following the C and D sequence ensures that the code is properly versioned, tested, and validated before it ever reaches a runtime environment.

NEW QUESTION: 42
What is a commit in Git?
A. A snapshot of all of the files in a project
B. A floating pointer to a stream of file changes
C. A fixed pointer that identifies the changes to a file
D. A list of files with the changes made to each file over time
Answer: (SHOW ANSWER)
When working with Guidewire Cloud Platform (GWCP), developers use Git for version control.
Understanding the internal mechanics of Git is essential for managing InsuranceSuite configuration changes.
A common misconception is that Git stores " diffs " or just the changes made to files. However, according to the Developing with Guidewire Cloud training, a commit is fundamentally a snapshot of the entire project at a specific point in time.
When you perform a commit, Git takes a " picture " of what all your files look like at that moment. To stay efficient, if a file has not changed, Git doesn ' t store the file again; instead, it stores a link to the previous identical version it has already stored. This snapshot includes metadata such as the author, the timestamp, and a reference to the " parent " commit that came before it. This allows Git to reconstruct the entire state of the configuration at any point in history.
Option C is incorrect because it describes a pointer to changes (a delta), which is how older version control systems like SVN worked. Option B is more descriptive of a " Branch, " which is a moving pointer to a commit. Option D describes the " History " or " Log " view. By treating every commit as a complete snapshot, Git ensures that the integrity of the Guidewire metadata is maintained, even when merging complex changes across different developer streams.

NEW QUESTION: 43
Given the following code sample:
var newBundle = gw.transaction.Transaction.newBundle()
var targetCo = gw.api.database.Query.make(ABCompany)
targetCo.compare(ABCompany#Name, Equals, " Acme Brick Co. ")
var company = targetCo.select().AtMostOneRow
company.Notes = " TBD "
Following best practices, what two items should be changed to create a bundle and commit this data change to the database? (Select two)

A. End with newBundle.commit()
B. Add Notes to the bundle
C. Add targetCo to the bundle
D. Add company to the bundle
E. Add runWithNewBundle(\newBundle - > { to the first line
Answer: A,D (LEAVE A REPLY)
In Guidewire InsuranceSuite, Bundle Management is the core mechanism for managing database transactions.
When you retrieve an entity via a query, as seen in the code sample, that entity is in read-only mode. To modify it and persist those changes, the entity must be associated with a Bundle.
1. Adding the Entity to the Bundle (Option D)
The code sample retrieves a company object, but it is currently " read-only " because it was fetched outside of the newBundle context. To make the entity editable, you must explicitly add it to the bundle using the add() method:
company = newBundle.add(company)
company.Notes = " TBD "
By adding the entity to the bundle, Gosu creates a " writable " clone of the object. Any changes made to the properties of this specific instance are tracked by the bundle. Without this step, setting company.Notes = " TBD " would result in a runtime exception stating that the entity is read-only.
2. Committing the Changes (Option A)
A bundle acts as a temporary " staging area " for changes. Simply modifying an object within a bundle does not automatically update the database. To persist the data, the developer must explicitly call the commit method:
newBundle.commit()
This triggers the database transaction, executing the necessary SQL UPDATE statements and clearing the bundle ' s state upon success.
Why other options are incorrect: * Option E describes the syntax for a runWithNewBundle block. While using runWithNewBundle is considered a best practice because it handles the commit and exception logic automatically, the question specifically asks what needs to be changed in the provided procedural code.
* Option B and C are incorrect because you do not add " Notes " (a property) or a " Query " object to a bundle; you only add Entities that you intend to modify or create.

NEW QUESTION: 44
Which statement is correct and recommended for writing GUnit tests?
A. Use the init() method to set up objects shared by all tests in a test class
B. Handle any exceptions thrown by test methods in the finally() method
C. Clear all instance variables of completed test in the tearDown() method
D. Use fluent assertions over conventional assert statements
Answer: A (LEAVE A REPLY)
GUnit is the Guidewire-specific testing framework based on JUnit, used to verify that Gosu classes and business rules function correctly. Efficient test writing requires a clear understanding of the test lifecycle, specifically how to manage resources and test data.
According to the Guidewire " System Health & Quality " training, the init() method (or equivalent
@BeforeClass setup logic in newer versions) is the recommended location for initializing resources that are expensive to create or are shared across all test methods within a specific class (Option A). By setting up shared objects-such as mock configuration data or static helper instances-in the init() phase, the developer ensures that the test suite runs faster and avoids redundant processing for every individual test case.

While Option C (clearing variables in tearDown()) is a valid memory management practice in some long- running Java environments, the primary focus of Guidewire GUnit training regarding the test lifecycle emphasizes the setup phase to ensure a consistent " known state " before tests execute. Option B is incorrect because GUnit is designed to catch and report exceptions as test failures; wrapping them in a manual finally block would obscure the failure and bypass the framework ' s reporting capabilities. Option D mentions fluent assertions; while modern and readable, conventional assertTrue, assertEquals, and assertNotNull remain the standard recommended assertion types in the core Guidewire Developer training curriculum.

NEW QUESTION: 45
An insurer has a number of employees who are working remotely from different locations. Displaying the employee ' s name in a drop-down list in the user interface must include the employee ' s location along with it. For example: John Smith London, UK; William Andy Dublin, Ireland; Eric Andy BC, Canada. How can a developer satisfy this requirement following best practices?
A. Define an entity name that concatenates the name fields and work locations.
B. Create a setter property in a Name enhancement class.
C. Create a displaykey that concatenates the name fields and work locations.
D. Enable Post On Change for name fields to modify how the name is displayed when referenced.
Answer: (SHOW ANSWER)
In the Guidewire InsuranceSuite data model, an entity ' s " Display Name " is the primary string representation used by the system whenever that entity is shown in the user interface-most notably in dropdowns (RangeInputs), search results, and labels. This is controlled via Entity Name Configuration (typically .en files or specialized Gosu classes).
According to the Data Model Architecture and InsuranceSuite Developer Fundamentals, when business requirements demand that users distinguish between entities with similar names (like " William Andy "), the best practice is to modify the entity ' s name definition to include differentiating data, such as a location or an ID. By defining an entity name that concatenates the employee ' s name fields with their work location, the developer creates a globally consistent identity for the object. This ensures that every time the " Employee " entity is selected from a dropdown, the user sees both the name and the location, reducing the risk of administrative error.
Using a Displaykey (Option C) is incorrect because displaykeys are meant for localized static text, not for dynamic data-driven identity logic. Post On Change (Option D) is a UI-level mechanism used to trigger server-side logic when a field value changes; it does not govern how an entity is fundamentally represented as a string. By modifying the entity name directly, the developer follows the " DRY " (Don ' t Repeat Yourself) principle, as the concatenation logic is written once at the data model level and automatically inherited by every PCF that references the entity.

NEW QUESTION: 46
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 this task?
B. Create an EmployeeAddress_Ext.tti file and add a new typecode
alternate
C. Open the EmployeeAddress.tti and add a new typecode alternate
D. Create an EmployeeAddress.ttx file and add a new typecode
alternate_Ext
E. Create an EmployeeAddress.tix file and add a new typecode alternate_Ext
Answer: (SHOW ANSWER)

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 newtypecodeadded 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.

NEW QUESTION: 47
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 (LEAVE A REPLY)
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 theDebuglog 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) shouldneverbe 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.

NEW QUESTION: 48
The Panel Ref in the screenshot below displays a List View with a toolbar. Add and Remove buttons have been added to the toolbar, but they appear in red, indicating an error. The Row Iterator has toAdd and toRemove buttons correctly defined.

What needs to be configured to fix the error?
A. Set the toCrealeAndAdd property of the row iterator
B. Sel the addVisible and removeVisible properties of the Add and Remove buttons
C. Set the iterator property of the Add and Remove buttons
D. Set the Visible property of the row iterator
Answer: C (LEAVE A REPLY)
In the GuidewirePage Configuration Framework (PCF), there is a strict functional relationship between toolbar buttons and the data they manipulate. When dealing withList Views (LVs), the "Add" and "Remove" buttons are specialized widgets known asIterator Buttons.
According to theInsuranceSuite Developer Fundamentalscurriculum, placing an Iterator Button in a toolbar is only the first step. For the button to be valid, it must be linked to a specificRow Iteratorlocated within the List View. This is accomplished by setting theiteratorproperty on the Add or Remove button to theIDof the target Row Iterator.
The red error in Guidewire Studio signifies a metadata validation failure. Even if the Row Iterator has the correct toAdd and toRemove logic defined (the "how" of the operation), the buttons themselves do not yet know "where" that logic resides. By setting the iterator property, you create a direct reference that tells the button which array of objects it is responsible for managing.
Why other options are incorrect:
* Option A:toCreateAndAdd is an optional property of the Row Iterator used for overriding the default object creation logic; it does not resolve the connection error between the button and the iterator.

* Option B:addVisible and removeVisible are boolean expressions used to hide buttons based on user permissions or object state; they do not fix structural metadata errors.
* Option D:The Visible property on an iterator affects whether the list is rendered, not whether the toolbar buttons are correctly linked.
Linking the button to the iterator ID is a fundamental best practice that ensures the UI remains synchronized with the underlying data bundle.

NEW QUESTION: 49
An insurer would like to include the Law Firm Specialty as part of the Law Firm ' s name whenever the name is displayed in a single widget. Which configurations follow best practices to meet this requirement?
A. Modify the Law Firm entity ' s displayname property to include the law firm ' s specialty.
B. Implement a getter method on the entity to return a formatted name that includes the law firm ' s specialty.
C. Place a Text Input widget in the ListView ' s Row container for the law firm ' s specialty.
D. Configure the entity name for the Law Firm entity to include law firm ' s specialty.
E. Add a custom field to the entity to store a concatenated display string.
F. Use a dynamic field to generate the display string to include the law firm ' s specialty.
G. Place a Text Cell widget in the ListView ' s Row container for the law firm ' s specialty.
Answer: D (LEAVE A REPLY)
In Guidewire InsuranceSuite, the standard and most efficient way to define how an object identifies itself visually across the entire application is by using Entity Names. This is a declarative configuration found in the metadata layer (specifically within .en files).
1. The Centralized Approach (Option D)
According to the InsuranceSuite Developer Fundamentals course, whenever a requirement asks for a consistent display format across " every widget " or " anywhere the name is displayed, " developers should use Entity Name configuration. By modifying the EntityName metadata for the Law Firm entity, you can define a template that concatenates the firm ' s name with its specialty (e.g., Name + " (" + Specialty + ") ").
This approach is considered a best practice for several reasons:
* Consistency: It ensures that every dropdown, list view, and detail view automatically displays the firm in the correct format without needing to modify hundreds of individual PCF files.
* Maintenance: If the business logic changes (e.g., they want to add the City instead of the Specialty), the change is made in exactly one place.
* Performance: Entity Names are handled efficiently by the platform ' s display engine, avoiding the overhead of custom Gosu calculations every time a widget renders.
2. Why Other Options are Discouraged
* Option B (Getter Method): While implementing a getter works, it requires you to manually point every single widget to this new property (e.g., LawFirm.FullDisplayName_Ext) instead of just using the standard entity reference.
* Options C and G: These only solve the problem for a single List View. They do not address the requirement to show the combined information " whenever the name is displayed " in other parts of the UI, such as Detail Views or search results.
* Option E (Custom Field): Storing a concatenated string in the database is a data redundancy anti- pattern. It creates extra storage overhead and requires complex logic to keep the concatenated string in sync whenever the Name or Specialty changes.
By utilizing the Entity Name configuration, developers leverage the Guidewire platform ' s built-in " stringify
" logic, which is the architecturally sound way to manage entity identity in the UI.

NEW QUESTION: 50
What type of Assessment Check ensures that applications have monitoring and logging frameworks in place?
A. Upgrades

B. Operations
C. Security
D. Performance
Answer: B (LEAVE A REPLY)

NEW QUESTION: 51
Succeed Insurance is developing multiple policy lines of business (LOB). The LOBs they are implementing are Homeowners (HO), Commercial Auto (CA), and Personal Auto (PA). They want to show key data elements of these LOBs on the exposure screen in ClaimCenter. To support this, you will need to modify the ExposureDetailDV container to support the different LOBs. Following best practices, which of the following implementations should be used?
A. Modify the ExposureDetailDV and add inline InputSets for LOB data needed for each region. Then use visibility logic to show and hide the LOB specific InputSets so each LOB only sees the data they need.
B. Create an InputSet for each LOB (e.g., LOBHOInputSet, LOBCAInputSet, and LOBPAInputSet).
Modify the ExposureDetailDV and add references to all InputSets, using visibility logic to show/hide them.
C. Create a set of LOB InputSets with the following modes: HO, CA, PA, and Default. Modify the ExposureDetailDV to add a reference to the new LOB InputSet, specifying the LOB value as the mode so the correct InputSet is displayed.
D. Create a single InputSet for all LOBs. Within the InputSet, specify visibility logic for each individual field based on the LOB the field is used for, then reference this InputSet in the ExposureDetailDV.
Answer: C (LEAVE A REPLY)
In Guidewire InsuranceSuite, when a single location must display significantly different content based on a specific property-such as a Line of Business (LOB)-the recommended architectural pattern is to use PCF Modes. While visibility logic (Option A or B) can technically hide or show elements, it leads to " bloated " PCF files that are difficult to maintain and performance-heavy, as the system must evaluate complex Boolean expressions for every element in the container.
Using Modes allows a developer to create multiple versions of an InputSet (or other containers) that share the same name but have different " Mode " identities (e.g., HO, CA, PA). In the parent configuration file (ExposureDetailDV), the developer adds a single InputSetRef. Instead of pointing to a static file, the def attribute points to the shared name of the InputSet, and the mode attribute is set to a dynamic expression, such as Exposure.Claim.LOBCode. At runtime, Guidewire ' s UI engine evaluates this expression and " swaps in " the specific PCF that matches the mode. If no specific mode matches, the system gracefully falls back to the Default mode. This approach promotes modularity, encapsulates LOB-specific logic within separate files, and ensures that the main ExposureDetailDV remains clean and readable. This is a core tenet of PCF Architecture and Dynamic UI management within the InsuranceSuite Developer curriculum.

NEW QUESTION: 52
Which GUnit base class is used for tests that involve Gosu queries in PolicyCenter?
A. GUnitTestClassBase
B. SuiteDBTestClassBase
C. PCUnitTestClassBase
D. PCServerTestClassBase
Answer: D (LEAVE A REPLY)
In theGuidewire System Health & Qualitytraining, understanding the hierarchy of GUnit base classes is essential for writing effective automated tests. While GUnitTestClassBase (Option A) provides basic testing functionality, it does not necessarily initialize the full application server environment or the database connection required for complex operations. For tests that require thefull Guidewire stack-including the ability to executeGosu

queriesagainst the database or interact with the bundle-developers must usePCServerTestClassBase(Option D) in PolicyCenter (or CCServerTestClassBase in ClaimCenter).
This base class ensures that:
* The Guidewire Application Server environment is "mocked" or started.
* The current user session is authenticated.
* The database transaction manager (Bundles) is available for queries and commits.
Using a lower-level base class for a query-based test would result in a NullPointerException or a NoSessionException because the Query API requires an active server context to translate Gosu into SQL.

NEW QUESTION: 53
An insurer with a self-managed InsuranceSuite implementation is preparing to transition to Guidewire Cloud Platform (GWCP). Which two Cloud Delivery Standards must be met before deployment? (Select two)
A. Performance tests must be developed and run for all functionality before an upgrade to the Cloud.
B. All new typelist and entity extension names include a three-character customer-specific suffix.
C. Database Consistency Check data issues that prevent upgrades must be fixed.
D. Customers must be on the most current General Availability (GA) version of the product being deployed to the Cloud.
Answer: (SHOW ANSWER)
The transition from a self-managed (on-premise) environment to Guidewire Cloud Platform (GWCP) involves a rigorous set of " Cloud Delivery Standards " designed to ensure the stability and supportability of the SaaS environment.
The first critical standard (Option C) involves the Database Consistency Checks (DCC). Before a database can be migrated to the cloud, it must be " clean. " Any data integrity issues-such as orphaned rows, invalid typecodes, or violated foreign key constraints-that are flagged by the DCC tool must be remediated. If these issues are not fixed, the automated migration scripts used during the cloud transition may fail, or the application may exhibit unpredictable behavior in the cloud environment.
The second standard (Option D) requires that the customer be on a supported General Availability (GA) version. Guidewire Cloud is built on a " Continuous Delivery " model, and the migration path is significantly streamlined when the source application is on a recent, stable version of the software. Attempting to move a legacy version that is several years out of date often requires multiple " hop " upgrades before the cloud transition can even begin.
While custom suffixes (Option B) and performance tests (Option A) are important parts of a project lifecycle, they are not the specific, mandatory " blocking " standards for the initial architectural transition in the same way that data integrity and product versioning are. Adhering to these standards minimizes risk and aligns the customer with the Guidewire SurePath methodology for cloud success.

NEW QUESTION: 54
The Panel Ref in the screenshot below displays a List View with a toolbar. Add and Remove buttons have been added to the toolbar, but they appear in red, indicating an error. The Row Iterator has toAdd and toRemove buttons correctly defined.

What needs to be configured to fix the error?

A. Set the toCrealeAndAdd property of the row iterator

B. Sel the addVisible and removeVisible properties of the Add and Remove buttons

C. Set the iterator property of the Add and Remove buttons

D. Set the Visible property of the row iterator

Answer: C (LEAVE A REPLY)

In the Guidewire Page Configuration Framework (PCF), there is a strict functional relationship between toolbar buttons and the data they manipulate. When dealing with List Views (LVs), the " Add " and " Remove

" buttons are specialized widgets known as Iterator Buttons.

According to the InsuranceSuite Developer Fundamentals curriculum, placing an Iterator Button in a toolbar is only the first step. For the button to be valid, it must be linked to a specific Row Iterator located within the List View. This is accomplished by setting the iterator property on the Add or Remove button to the ID of the target Row Iterator.

The red error in Guidewire Studio signifies a metadata validation failure. Even if the Row Iterator has the correct toAdd and toRemove logic defined (the " how " of the operation), the buttons themselves do not yet know " where " that logic resides. By setting the iterator property, you create a direct reference that tells the button which array of objects it is responsible for managing.

Why other options are incorrect:

* Option A: toCreateAndAdd is an optional property of the Row Iterator used for overriding the default object creation logic; it does not resolve the connection error between the button and the iterator.

* Option B: addVisible and removeVisible are boolean expressions used to hide buttons based on user permissions or object state; they do not fix structural metadata errors.

* Option D: The Visible property on an iterator affects whether the list is rendered, not whether the toolbar buttons are correctly linked.

Linking the button to the iterator ID is a fundamental best practice that ensures the UI remains synchronized with the underlying data bundle.

NEW QUESTION: 55

Given the following code example:

var query = gw.api.database.Query.make(Claim)

query.compare(Claim#ClaimNumber, Equals, " 123-45-6798 ")

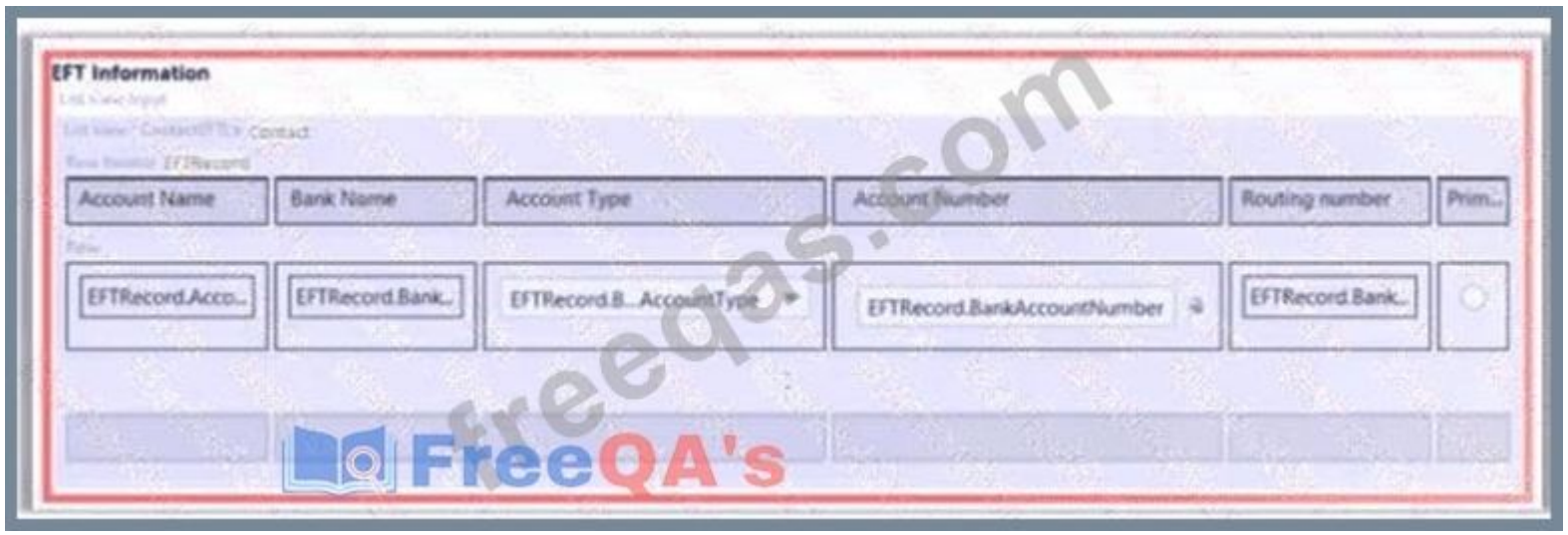
var claim = query.select().AtMostOneRow

According to best practices, which logic returns notes with the topic of denial and filters on the database?

A. var notesQuery = gw.api.database.Query.make(Note); var denialNotes = notesQuery.select().where(\elt

- > elt.Topic==NoteTopicType.TC_DENIAL)
B. var denialNotes = claim.Notes.where(\elt - > elt.Topic==NoteTopicType.TC_DENIAL)
C. var notesQuery = gw.api.database.Query.make(Note); notesQuery.compare(Note#Topic, Equals, NoteTopicType.TC_DENIAL); notesQuery.compare(Note#Claim, Equals, claim); var denialNotes = notesQuery.select()
D. var notesQuery = gw.api.database.Query.make(Note); notesQuery.compare(Note#Topic, Equals, NoteTopicType.TC_DENIAL); var denialNotes = notesQuery.select()
Answer: C (LEAVE A REPLY)
Efficiency in Guidewire performance relies heavily on the " Database-First " principle. To fulfill the requirement of filtering notes by both Claim and Topic specifically on the database, a new query must be constructed using the Query API.
Option C is the only correct answer because it uses the .compare() method to apply two specific filters:
* Topic Filter: It filters for the specific typecode TC_DENIAL.
* Claim Filter: It links the query to the specific claim object found in the previous step.
By setting these parameters before calling .select(), Guidewire generates a single SQL statement: SELECT * FROM cc_note WHERE topic = ' denial ' AND claimid = ... The database performs the heavy lifting and returns only the relevant records.
Options A and B are anti-patterns. They fetch all notes (Option B) or execute a broad query (Option A) and then use the Gosu .where() method to filter in the application server ' s memory. This is highly inefficient.
Option D is incomplete as it would return every denial note in the entire system, regardless of which claim it belongs to.

NEW QUESTION: 56
ContactManager provides an inline reference to an editable list view on the Contact Basics screen that supports adding and editing of banking information for contacts. The screenshot below shows this list view in Studio. There is an error within the red outline.



Which configuration changes are necessary to resolve the error? (Select two)
A. Add a toolbar widget to the list view input
B. Add and configure Iterator buttons
C. Replace the list view input with a panel ref
D. Replace the list view PCF with an inline list view
E. Add edit permissions to the row iterator
Answer: A,B (LEAVE A REPLY)
In the Guidewire Page Configuration Framework (PCF), displaying a list of data within a Detail View (DV) requires specific container widgets. When a developer uses a ListViewInput to embed an existing List View into a form, they are essentially creating an " editable grid " section.

1. The Requirement for a Toolbar (Option A)
According to the InsuranceSuite Developer Fundamentals guide, a ListViewInput is a specialized widget that acts as a wrapper for a List View. Unlike a standard List View displayed on its own page (which inherits the page ' s toolbar), a ListViewInput exists inside a Detail View column. For the list to be interactive-allowing users to add new bank accounts or remove existing ones-the ListViewInput must have its own Toolbar. In Guidewire Studio, if a ListViewInput is marked as editable but lacks a toolbar, the metadata validator will flag an error because there is no container to hold the necessary action buttons.
2. Configuring Iterator Buttons (Option B)
Once the Toolbar is added to the ListViewInput, it remains empty until Iterator Buttons are placed inside it.
These buttons (typically the " Add " and " Remove " buttons) must be explicitly configured to point to the Row Iterator defined within the referenced List View PCF.
The error in the screenshot is resolved by:
* Selecting the ListViewInput in the PCF tree.
* Adding a Toolbar child widget.
* Adding Iterator Buttons (Add/Remove) to that toolbar.
* Linking those buttons to the correct iterator ID.
This combination provides the end-user with the UI controls needed to manipulate the banking information array. Options C and D represent alternative ways to structure the UI but do not address the specific configuration error of the ListViewInput widget. Option E relates to security and runtime behavior, not the structural metadata requirements of the PCF layout engine.

NEW QUESTION: 57
The Officials list view in ClaimCenter displays information about an official called to the scene of a loss (for example, police, fire department, ambulance). The base product captures and displays only three fields for officials. An insurer has added additional fields but still only displays three fields. The insurer has requested a way to edit a single record in the list view to view and edit all of the officials fields. Which location type can be used to satisfy this requirement?
A. Forward
B. Page
C. Popup
D. Location group
Answer: (SHOW ANSWER)
In Guidewire InsuranceSuite UI design, balancing information density is a common challenge.List Views (LVs)are optimized for showing multiple records at once but are limited by horizontal screen real estate.
When an entity has more fields than can comfortably fit in a table-as is the case with the expanded
"Officials" entity-Guidewire best practices recommend using aPopup(Option C) for detailed editing.
A Popup is a specializedLocationtype that opens a secondary window over the current page. This allows the developer to embed a fullDetail View (DV)containing all the new fields (police badge numbers, department contact info, etc.) without navigating the user away from the main Claim screen.
This "List-Detail" pattern is typically implemented by making one of the fields in the List View (like the Official's name) aLinkor by adding an "Edit" button that calls the popover or push method to launch the Popup.
Other location types are inappropriate for this specific requirement. AForward(Option A) is a non-visual location used for logical branching (deciding where to send a user based on data). APage(Option B) would take the user completely away from the current context, which is disruptive for a simple edit. ALocation Group(Option D) is used for structural navigation in the sidebar, not for individual record interaction. By utilizing a Popup, the developer provides a focused, high-density editing environment that maintains the user's workflow within the ClaimCenter application.

NEW QUESTION: 58
You have created a list view file BankAccountsLV that will display a list of bank accounts. You have added a Toolbar and Iterator Buttons, but when you try to select the Iterator related to the Iterator Buttons, the list of available Iterators is empty.
What is needed to fix this problem?
A. In the BankAccountsLV file, click on the Row, select the "Exposes" tab, click the "+", select 'Expose Iterator', and select the iterator defined in BankAccountsLV.
B. Manually enter the Iterator name of BankAccountsLV, and Studio will find the file.
C. In the BankAccountsLV file, click on the Row Iterator, select the "Exposes" tab, click the "+", select "Expose Iterator", and select the iterator defined in BankAccountsLV.
D. In the BankAccountsLV file -> "Exposes" tab, click the "+", select "Expose Iterator", and select the iterator defined in BankAccountsLV.
E. Replace the Iterator Buttons with separate Toolbar Buttons to "Add" and "Remove" rows from the Iterator.
F. Open the BankAccountsLV file and from the top menu select "Build -> Recompile BankAccountsLV"
Answer: C (LEAVE A REPLY)
In the Guidewire Page Configuration Framework (PCF), communication between widgets is strictly governed by visibility and scope. A common scenario involves usingIterator Buttons(Add/Remove) within a toolbar to manipulate a list of data. These buttons must be explicitly linked to aRow Iteratorwidget to know which collection of data they should act upon.
The issue described-where the "Iterator" dropdown is empty when configuring the buttons-is a result of the Iterator's properties not being "exposed" to the containing page. In Guidewire Studio, widgets within a PCF file (like an LV) are not automatically visible to the external pages that call them.
To make an internal widget like a Row Iterator accessible to a parent container (such as a Detail View panel or a Screen where the toolbar resides), the developer must use theExposestab.
According to best practices, the developer should select theRow Iteratorelement in the BankAccountsLV file, navigate to theExposestab, and add an entry for "Expose Iterator." This creates a reference that allows the PCF editor to "see" the iterator. Once this configuration is saved, the Iterator Buttons on the calling page will find the named iterator in the dropdown menu. Options A, B, and D are incorrect because they target the wrong level of the PCF hierarchy or suggest manual entry which the Studio UI does not support for this specific linkage. Option E is a workaround that bypasses the built-in functionality of Iterator Buttons, and Option F is a general maintenance step that does not resolve metadata configuration issues.

NEW QUESTION: 59
Succeed Insurance needs to extend the contact functionality to support tracking agency information. The new agency entity should have all of the fields of ABCompany, but include fields that are specific to the agency.
Following best practices, which of the following options would implement this requirement?
A. A new foreign key should be added to ABCompany that points to a new Agency_Ext entity. The new fields should be added to the new Agency entity.
B. A new Agency_Ext entity should be added so that ABCompany becomes a subtype of Agency_Ext.
The new fields should be added to the new Agency entity.
C. A new array should be added to ABCompany that points to a new Agency_Ext entity. The new fields should be added to the new Agency_Ext entity, including a foreign key pointing back to ABCompany.
D. Add a new Agency subtype of ABCompany. The new fields should be added to the new Agency_Ext subtype.
Answer: D (LEAVE A REPLY)
The Guidewire data model is designed to support Subtyping, which is a powerful mechanism for creating specialized versions of existing entities. This is specifically used within the Contact and Company hierarchies.

When a requirement states that a new entity must have all the fields of an existing entity (ABCompany) plus additional specific fields, a subtype is the correct architectural choice.
By creating an Agency subtype of ABCompany, the new entity automatically inherits all the metadata, fields, and relationships defined on the parent ABCompany and its ancestor, ABContact. The developer then adds the agency-specific fields (such as " Agency License Number ") directly to the subtype. This " is-a " relationship is much more efficient than using a Foreign Key (Option A) or an Array (Option C), which are intended for " has-a " relationships.
Subtyping ensures that the Agency records can still be treated as ABCompany or ABContact objects in Gosu logic and UI components (like search pages), while still allowing for specialized behavior and data storage.
Following naming conventions, these custom subtypes often include the _Ext suffix to distinguish them from out-of-the-box subtypes. This approach minimizes data redundancy and leverages the built-in polymorphic capabilities of the InsuranceSuite Data Model, ensuring that the system remains scalable and easy to maintain during future upgrades.

NEW QUESTION: 60
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 (LEAVE A REPLY)
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.

NEW QUESTION: 61
A developer is creating an enhancement class for the entity AuditMethod_Ext in PolicyCenter for an insurer, Succeed Insurance. Which package structure of the gosu class and function name follows best practice?
A. si.pc.enhancements.entity, determineAuditType()
B. si.pc.enhancements.entity, determineAuditType_Ext()
C. gw.job.audit, determineAuditType()

D. gw.entity.enhancement, determineAuditType_Ext()
Answer: B (LEAVE A REPLY)
Guidewire emphasizes a strict naming and packaging convention for custom Gosu classes and enhancements to ensure code clarity and to prevent "namespace collisions" during platform upgrades. For a customer like
"Succeed Insurance," the best practice is to use a unique prefix for the package structure, typically derived from the company's initials and the specific application.
In this case,"si.pc"(Succeed Insurance PolicyCenter) is the appropriate starting point for the package. Placing enhancements in a sub-package like"enhancements.entity"(Option B) logically organizes the code by its function, separating entity logic from other business rules or integration classes. This structure ensures that developers can easily locate custom logic added to both base entities and custom entities like AuditMethod_Ext.
Regarding the function name, Guidewire best practices for enhancements dictate that custom methods added to an entity should include the_Extsuffix (e.g., determineAuditType_Ext()). This is crucial because if Guidewire later releases a product update that adds a method with the same name (determineAuditType) to the base entity, the customer's version will not conflict with the base version.
Options C and D use the gw namespace, which is strictly reserved for Guidewire's internal "Out of the Box" code. Using the gw package for custom code can lead to severe compilation errors or unexpected behavior during upgrades, as the Guidewire platform assumes total ownership of that namespace. Therefore, utilizing the insurer's unique package prefix combined with the _Ext suffix on the method is the only approach that aligns with Guidewire's certification standards and long-term maintenance requirements.

NEW QUESTION: 62
A query is known to return 500,000 rows. Which two are recommended to process all 500,000 rows efficiently? (Select two)
A. Use Google Iterables.
B. Use setPageSize().
C. Use a batch process for large result sets.
D. Chunk the results into page sets.
E. Sort the result by entity name.
Answer: B,C (LEAVE A REPLY)
Processing extremely large datasets-such as 500,000 rows-presents significant challenges for memory management and transaction stability in Guidewire. To handle this efficiently, developers must use the Gosu Query API correctly and choose the right execution context.
The first best practice is the use of setPageSize() (Option B). When a query is executed, by default, the system might attempt to fetch a large number of rows into the application server ' s memory. By calling setPageSize (50) or setPageSize(100) on the query object, the developer instructs the database driver to fetch only a small
" page " of records at a time. This keeps the memory footprint of the Gosu bundle low and prevents OutOfMemory errors, even though the developer can still iterate through the entire 500,000-row result set as if it were a single collection.
The second best practice is to move such a heavy operation into a Batch Process (Option C). Executing a

500,000-row loop within a UI request or a standard rule would likely cause a web server timeout or block other threads. A Batch Process runs in the background, has its own dedicated work queue, and can be configured to " checkpoint " its progress. This means if the server restarts, the batch process can potentially resume where it left off.
Options like sorting (Option E) can actually hinder performance on large sets if the database index is not optimized for that sort. " Chunking " (Option D) is conceptually similar to paging, but setPageSize() is the specific, built-in method provided by the Guidewire Query API to achieve this.

NEW QUESTION: 63
An InsuranceSuite implementation project is preparing for deployment to the Guidewire Cloud Platform.
Which two Cloud Delivery Standards must be met before deployment? (Select two)
A. There are no instances of single statements with multiple expansion operators(*)
B. GUnit tests must be combined into suites and executed in Studio prior to each code deployment
C. The default system user su is configured as the second argument of the runWithNewBundle method
D. New entities and new columns added to existing entities use a customer suffix such as _Si
E. Log files contain no PII (Personally Identifiable Information) as clear text
Answer: (SHOW ANSWER)
Moving to the Guidewire Cloud Platform (GWCP) introduces a set of mandatory " Cloud Delivery Standards
" designed to ensure that customer implementations are secure, upgradeable, and performant. Two of the most critical pillars in these standards are Metadata Naming Conventions and Data Privacy/Security.
First, naming conventions (Option D) are essential for maintaining the " Upgrade-Safe " nature of the cloud.
In Guidewire Cloud, the base product is updated frequently. To prevent custom metadata (new entities or columns) from conflicting with future Guidewire base product releases, all extensions must use a unique customer suffix. While the standard example is _Ext, insurers often use their specific company initials, such as _Si for Succeed Insurance. This ensures that the custom data model remains distinct from the gw namespace.
Second, the Observability and Security standards strictly forbid the logging of Personally Identifiable Information (PII) in plain text (Option E). In the cloud, logs are aggregated and viewed via tools like CloudWatch or Kibana. If sensitive data like Social Security Numbers, Credit Card numbers, or personal addresses are logged as clear text, it constitutes a major security risk and a violation of compliance standards like GDPR or SOC2.
Guidewire ' s automated Quality Gates will often block a deployment if it detects potential PII leakage in the Gosu logging code.
Options A and B are general development practices but not the primary delivery standards that block deployment. Option C is actually a security risk; hardcoding the " su " (Super User) in bundle management is generally discouraged in favor of more granular permission handling.

NEW QUESTION: 64
Succeed Insurance has information that they want to display on multiple pages with the same layout. Which PCF container types can be used to meet this requirement? (Choose 3)
A. DetailView
B. Popup
C. LocationGroup
D. ListView
E. InputSet
F. Worksheet
Answer: (SHOW ANSWER)
One of the primary goals of PCF Configuration is modularity and reuse. Guidewire provides specific Container Widgets that allow developers to define a layout once and reference it in multiple locations throughout the application.

* DetailView (DV): This is the standard container for displaying field-value pairs, typically organized in columns. A DV can be defined as a standalone PCF file and then referenced on multiple pages (like a Policy File screen and a Claim Summary screen) using a DetailViewRef.
* ListView (LV): This container is used for tabular data or grids. Similar to DVs, LVs are often created as separate files so that the same table structure (with the same columns, sorting, and filtering) can be reused across different parts of the suite.
* InputSet: This is a more granular container used specifically within DetailViews. An InputSet allows a developer to group a set of related widgets (like an address block or a set of contact fields). By using an InputSetRef, a developer can ensure that the " Address Layout " is identical on every screen that requires it, drastically reducing maintenance time.
In contrast, Popups, LocationGroups, and Worksheets are classified as Locations, not containers. Locations define where a user is in the application (the URL or navigation state), whereas Containers (DV, LV, InputSet) define how the data is laid out inside those locations. Selecting these three containers allows for a " DRY " (Don ' t Repeat Yourself) development approach, which is a key best practice in InsuranceSuite Developer Fundamentals.

NEW QUESTION: 65
Succeed Insurance would like a list of all Notes related to all Policies for an Account. Which approach follows best practices for retrieving this data more efficiently?
A. Code snippetvar policyNotes = Query.make(Note).compare(Note#Policy, Relop.Equals, policy).
compare(Note#Account, Relop.Equals, account).select()
B. Code snippetvar policyNotes = account.Policies*.Notes.toList()
C. Code snippetvar policyNotes : ArrayList < Note > for(policy in account.Policies) {for (note in policy.
Notes) {policyNotes.add(note)}}
D. Code snippetvar policyNotes = account.Policies*.Notes*.DisplayName
Answer: (SHOW ANSWER)
In Guidewire InsuranceSuite, developers frequently need to " reach across " one-to-many relationships to collect data from nested arrays. In this scenario, the goal is to retrieve a flattened list of all Note entities associated with all Policy objects linked to a specific Account.
According to Advanced Gosu best practices, the most efficient and idiomatic way to handle this is by using the Expansion Operator (*). As shown in Option B, the syntax account.Policies*.Notes performs what is known as " collection flattening. " When the expansion operator is applied to the Policies array, Gosu understands that it should look at every policy in that collection and access the Notes array for each. It then automatically flattens these multiple sub-collections into a single, comprehensive list of Note objects. Calling .
toList() at the end ensures the result is captured in a standard, manipulatable collection format.
This approach is vastly superior to nested for loops (Option C). Manual iteration through nested arrays is a primary cause of the " N+1 " query problem and " Bundle Bloat. " In nested loops, the system may perform a separate database fetch for every policy and then another for every note, loading every single entity into the current transaction bundle, which consumes excessive memory and CPU time. The expansion operator, however, is highly optimized within the Gosu Runtime to handle these traversals more gracefully.
Option D is incorrect because it uses a second expansion operator to retrieve the DisplayName property, resulting in a list of Strings rather than a list of Note entities. Option A, while using the Query API, is logically disconnected from the root account object already in memory and represents a more complex search- based approach rather than a relationship-based retrieval. Therefore, the expansion operator is the verified standard for efficient, readable data collection in Gosu.

NEW QUESTION: 66
An insurer wants to add a new typecode for an alternate address to a base typelist EmployeeAddress that has not been extended. Following best practices, which step must a developer take to perform this task?

A. Create an EmployeeAddress_Ext.tti file and add a new typecode alternate.
B. Create an EmployeeAddress.tlx file and add a new typecode alternate_Ext.
C. Create an EmployeeAddress.ttx file and add a new typecode alternate_Ext.
D. Open the EmployeeAddress.tti and add a new typecode alternate.
Answer: (SHOW ANSWER)
Adding custom codes to an out-of-the-box typelist is a fundamental task in Data Model Configuration. To ensure that the configuration is upgrade-safe and follows Guidewire ' s architectural standards, developers must use the Typelist Extension mechanism.
The first rule is that base .tti (Typelist Internal) files must never be edited (ruling out Option D).
Modifications to these files will be lost during a platform upgrade. Instead, developers must use a .ttx (Typelist Extension) file. The filename must match the base typelist exactly (e.g., EmployeeAddress.ttx).
Adding _Ext to the filename itself (Option A) is incorrect and will prevent the application from merging the metadata correctly.
The second rule concerns the naming of the new typecode. According to the InsuranceSuite Developer standards, any code added by a customer to a Guidewire-owned typelist should include the _Ext suffix (e.g., alternate_Ext). This " namespacing " protects the customer from future collisions. If a future release of PolicyCenter or ClaimCenter includes a base alternate code in the EmployeeAddress typelist, the customer ' s version will remain distinct, preventing database errors or logic failures during the upgrade process.
By creating the .ttx file and using the _Ext suffix on the typecode (Option C), the developer adheres to the Open Type System principles. This ensures the custom data is properly recognized by the UI and Gosu rules while remaining perfectly aligned with the Guidewire Cloud Delivery Standards.