



Guidewire InsuranceSuite-Developer Dumps

Shared by Walton on 09-10-2026

For More Free Questions and Preparation Resources

Check the Links on Last Page



Question 1

Question Type: MultipleChoice

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?

Options:

- 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:

D

Explanation:

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.

Question 2

Question Type: MultipleChoice

An insurer wants to determine how many activities have been generated to review a denial decision with an Account Manager. A developer has written this query:

```
uses gw.api.database.Query
```

```
var queryResults = Query.make(Activity).select().toCollection().where(
```

```
\ act -> act.ActivityPattern.Code == "account_denial" )
```

This query does not perform as well as it should. Where should the results be filtered to follow best practices?

Options:

- A- On the application server
- B- In the application cache
- C- In a block statement
- D- In the database

Answer:

D

Explanation:

In Guidewire InsuranceSuite, the performance of an application is heavily dependent on how efficiently it interacts with the database. The Gosu Query API is designed to allow developers to retrieve exactly the data they need by translating Gosu logic into optimized SQL statements.

The developer's current approach in the code snippet is a significant anti-pattern. By calling `.select().toCollection()`, the system executes a broad SQL query that retrieves every single record from the Activity table and loads them into the application server's memory (the 'Bundle'). Only after these potentially thousands of records are in memory does the `.where()` block filter them. This causes high memory consumption, increased network latency, and unnecessary pressure on the application server's garbage collector.

According to the Gosu Queries lesson in the InsuranceSuite Developer Fundamentals course, the best practice is to filter results in the database (Option D). This is achieved by using the `compare()` or `join()` methods on the Query object before calling `.select()`. For example, using

query.compare(Activity#ActivityPattern, Equals, myPattern) ensures that the filtering is handled by the database engine via a SQL WHERE clause. Consequently, only the specific records that match the 'account_denial' criteria are sent over the network and loaded into memory. Following this practice minimizes the data 'payload,' reduces the execution time of the logic, and ensures the application remains scalable even as the volume of activities grows over time. Adhering to database-side filtering is a core requirement for passing Quality Gates in Guidewire Cloud environments.

=====

Question 3

Question Type: MultipleChoice

Given the following screen showing a DetailView in Guidewire Studio highlighted in red:



Which single item added directly to the detail view will correct the error shown, with no further errors?

Options:

- A- Add a row iterator to the detail view
- B- Add a toolbar to the detail view
- C- Add a list view to the detail view
- D- Add an input column to the detail view

Answer:

D

Explanation:

In Guidewire InsuranceSuite PCF (Page Configuration File) Configuration, the hierarchy and structural requirements of container widgets are strictly enforced by the Studio compiler and visual editor. A DetailView (DV) is a specific type of container designed to display and edit individual fields (atomic widgets) in a column-based layout.

When a developer adds a DetailView to a PCF, Studio will initially display it in red, indicating a validation error. This occurs because, according to the PCF Architecture standards, a DetailView is not a direct container for input widgets like TextInput or RangeInput. Instead, it requires a layout-specific child element to define how those widgets are organized. The mandatory child for a standard DetailView is the InputColumn.

An InputColumn provides the necessary structure to align labels and their corresponding input widgets. Without at least one InputColumn defined directly under the DetailView, the DV is considered incomplete and structurally invalid. Adding an InputColumn (Option D) satisfies the minimum container requirement, effectively clearing the validation error and allowing the developer to then place field-level widgets within that column.

Regarding the other options: A RowIterator (Option A) is a component specifically used within a ListView to iterate over a collection of data and is not a direct child of a DetailView. While a ListView (Option C) can be displayed within a DV, it must be wrapped in a ListViewInput or placed inside an InputSet, and simply adding a 'list view' does not satisfy the core structural requirement for an input-based DV layout as directly as an InputColumn. A Toolbar (Option B) is typically associated with a Screen, PanelRef, or ListView, rather than being a structural correction for a DetailView. Therefore, the InputColumn is the fundamental architectural element required to make the DV valid.

=====

Question 4

Question Type: MultipleChoice

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.

The screenshot shows a form titled "Additional Info" with the following fields:

- Medical License: A text input field.
- Medical Specialty: A dropdown menu with "<none>" selected.
- Preferred Vendor?: Radio buttons for "Yes" and "No", with "No" selected.
- Affiliated Hospital: A dropdown menu with "Health South" selected. This field is highlighted with a red rectangular box.
- Hospital Address: A text area containing "9032 Market St", "San Francisco", and "California".

What is the appropriate configuration approach to satisfy these requirements?

Options:

- 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

Explanation:

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.

=====

Question 5

Question Type: MultipleChoice

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 snippet

```
var policyNotes = Query.make(Note)
    .compare(Note#Policy, Relop.Equals, policy)
    .compare(Note#Account, Relop.Equals, account)
    .select()
```

B.

Code snippet

```
var policyNotes = account.Policies*.Notes.toList()
```

C.

Code snippet

```
var policyNotes : ArrayList
for(policy in account.Policies) {
    for (note in policy.Notes) {
        policyNotes.add(note)
    }
}
```

```
}
```

```
}
```

D.

Code snippet

```
var policyNotes = account.Policies*.Notes*.DisplayName
```

Options:

A- Option A

B- Option B

C- Option C

D- Option D

Answer:

B

Explanation:

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

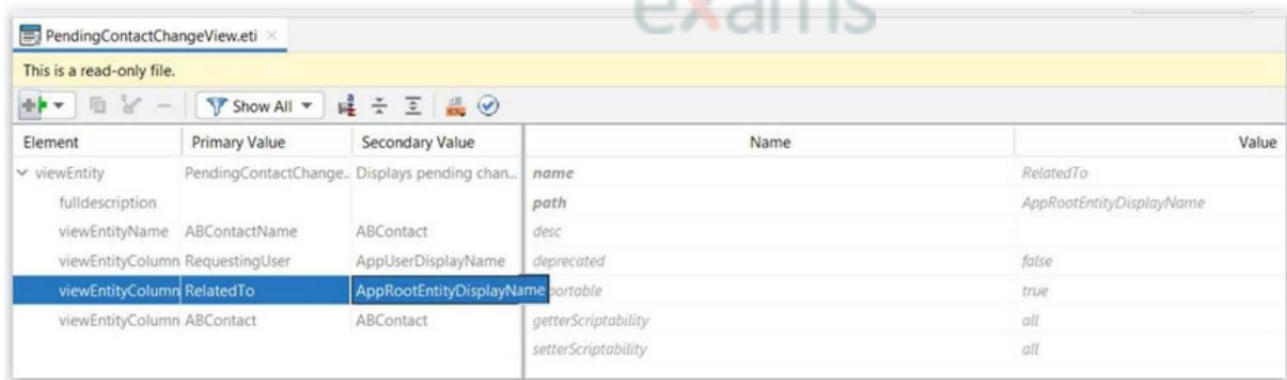
represents a more complex search-based approach rather than a relationship-based retrieval. Therefore, the expansion operator is the verified standard for efficient, readable data collection in Gosu.

=====

Question 6

Question Type: MultipleChoice

What configuration item is needed to add ABContact.Notes to PendingContactChangeView following best practices?



Element	Primary Value	Secondary Value	Name	Value
viewEntity	PendingContactChange..	Displays pending chan..	name	RelatedTo
fulldescription			path	AppRootEntityDisplayName
viewEntityName	ABContactName	ABContact	desc	
viewEntityColumn RequestingUser		AppUserDisplayName	deprecated	false
viewEntityColumn RelatedTo		AppRootEntityDisplayName	portable	true
viewEntityColumn ABContact		ABContact	getterScriptability	all
			setterScriptability	all

Options:

- 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:

A

Explanation:

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.

=====

Question 7

Question Type: MultipleChoice

Succeed Insurance would like to count the number of High Priority Activities that are related to a Job. Which approach follows best practices to meet this requirement?

Options:

- A- `var count = 0; job.Activities.each(\act -> act.Priority == TC_HIGH) { count += 1 }; print(String.format('Count of High Priority Activities: %s', {count}))`
- B- `var actList = job.Activities.toList(); var count = actList.countWhere(\act -> act.Priority == TC_HIGH); print(String.format('Count of High Priority Activities: %s', {count}))`
- C- `var count = job.Activities.countWhere(\act -> act.Priority == TC_HIGH); print(String.format('Count of High Priority Activities: %s', {count}))`
- D- `var count = 0; for (act in job.Activities) { if (act.Priority == TC_HIGH) { count += 1 } }; print(String.format('Count of High Priority Activities: %s', {count}))`
- E- `var count = 0; job.Activities.where(\act -> act.Priority == TC_HIGH) { count += 1 }; print(String.format('Count of High Priority Activities: %s', {count}))`

Answer:

C

Explanation:

In Guidewire InsuranceSuite, when working with entity arrays (like `job.Activities`), the Gosu language provides a powerful set of Collection Enhancements that allow developers to perform operations on data sets with minimal code and maximum readability. These enhancements are built on top of standard Java collections but offer a more functional programming approach.

Option C is the correct best practice because it utilizes the `countWhere` enhancement. This method takes a block (a lambda expression) that defines a predicate and returns an integer representing the count of elements that satisfy that condition. This approach is superior to manual iteration for several reasons. First, it is declarative; it tells the system what to do (count where priority is high) rather than how to do it (initialize a counter, loop, check if, increment). This reduces the likelihood of 'boilerplate' coding errors, such as off-by-one errors or scope issues with the counter variable.

Comparing this to the other options: Option D uses a traditional for loop, which is functional but overly verbose and less idiomatic in Gosu. Option B is inefficient as it creates an entirely new List object in memory (`toList()`) before counting, which is unnecessary overhead. Option A incorrectly uses `each`, which is intended for side effects, and Option E incorrectly uses `where`, which is intended for filtering a subset of objects into a new collection, not for executing a counting block.

By using `countWhere` directly on the entity array, the developer writes cleaner, more maintainable code that aligns with the Gosu Coding Standards taught in the Advanced Gosu curriculum. This method ensures that the logic is encapsulated in a single, readable line, making it easier for other developers to understand the business intent of the logic.

=====

P2P
exams

Question 8

Question Type: MultipleChoice

Given the following query:

```
uses gw.api.database.Query
```

```
var query = Query.make(Claim)
```

```
query.compare(Claim#ClaimNumber, Equals, "123-45-6789")
```

```
var claim = query.select().AtMostOneRow
```

Which follows the best practice to find the urgent open activities of the claim, considering the memory usage and bundle size?

A.

```
var activityQuery = gw.api.database.Query.make(Activity)
activityQuery.compare(Activity#Status, Equals, TC_OPEN)
activityQuery.compare(Activity#Claim, Equals, claim)
var urgentActivities = activityQuery.select().where(\ activity -> activity.Priority == TC_URGENT)
```

B.

```
var urgentActivities = claim.Activities.where(\ activity -> activity.Priority == Priority.TC_URGENT
and activity.Status == TC_OPEN)
```

C.

```
var urgentActivities = claim.Activities.where(\ activity -> activity.Status == TC_OPEN)
.where(\ activity -> activity.Priority == TC_URGENT)
```

D.

```
var activityQuery = gw.api.database.Query.make(Activity)
activityQuery.compare(Activity#Priority, Equals, Priority.TC_URGENT)
activityQuery.compare(Activity#Status, Equals, TC_OPEN)
activityQuery.compare(Activity#Claim, Equals, claim)
var urgentActivities = activityQuery.select()
```

Options:

- A- Option A
- B- Option B
- C- Option C
- D- Option D

Answer:

D

Explanation:

In Guidewire InsuranceSuite, managing how data is retrieved from the database is critical for system performance, specifically regarding memory usage and the bundle size. A primary goal for any developer is to ensure that filtering happens at the database level rather than within the application server's memory.

Options B and C demonstrate a common but inefficient pattern: accessing an array directly (e.g., `claim.Activities`). When you access an array on an entity, the Guidewire platform automatically loads every related record in that array into the application server's memory and adds them to the current Bundle. If a claim has hundreds of activities, but you only need the three that are 'Urgent' and 'Open,' Options B and C still force the system to load all of them. This consumes significant memory and increases the overhead of the bundle, which can lead to performance degradation or 'Out of Memory' errors in high-volume environments.

Option D is the verified best practice. By using the Gosu Query API (`Query.make(Activity)`), the developer can build a specific SQL statement. Using the `.compare()` method for the Priority, Status, and the link to the Claim ensures that all three criteria are passed to the database as part of the WHERE clause. When `.select()` is called, the database engine filters the records and returns only the specific rows that meet all requirements. Consequently, only the necessary objects are loaded into the application server's memory and added to the bundle. Option A is less efficient because it uses a Gosu lambda (`.where`) after the select, which performs the final filtering in memory rather than at the database tier. Following the pattern in Option D minimizes the data 'payload' and ensures the application remains scalable and responsive.

=====



To Get Premium Files for InsuranceSuite-Developer Visit

<https://www.p2pexams.com/products/insurancesuite-developer>

For More Free Questions Visit

<https://www.p2pexams.com/guidewire/pdf/insurancesuite-developer>

20%
DISCOUNT

P2P
exams