Up-to-date Questions and Answers from authentic resources to improve knowledge and pass the exam at very first attempt. ----- Guaranteed.

KILL EXAMS

ACD201 MCQs
ACD201 TestPrep
ACD201 Study Guide
ACD201 Practice Test
ACD201 Exam Questions

PASS

killexams.com

Appian

ACD201

Appian Certified Senior Developer - 202

ORDER FULL VERSION

https://killexams.com/pass4sure/exam-detail/ACD201

Question: 470

In a banking application, a Custom Data Type named AccountTransaction is used to store transaction details, including a nested CDT for audit logs. The application requires that audit logs be retained for compliance but not loaded into memory for every transaction query to optimize performance. You decide to separate the audit logs into a different database table. Which approach best implements this separation while maintaining referential integrity?

A. Create a new CDT for audit logs and link it to AccountTransaction using a foreign key
B. Store audit logs in a separate data store and manage relationships via process models
C. Use a database view to combine AccountTransaction and audit logs for queries
D. Flatten the audit logs into the AccountTransaction CDT with a JSON field

Answer: A

Explanation: Creating a new CDT for audit logs and linking it with a foreign key ensures referential integrity and allows separate storage, reducing memory usage during transaction queries. Storing audit logs in a separate data store complicates relationships and maintenance. A database view is not suitable for separating data storage. Flattening audit logs into a JSON field sacrifices structure and query efficiency.

Question: 471

An Appian application is experiencing user-facing performance issues, with users reporting that a task form takes 10 seconds to load. The form includes a complex SAIL interface with multiple a!queryEntity() calls to fetch data from a database with millions of rows. The Process Monitoring tab shows high memory usage for the process model, and the application server logs indicate frequent garbage collection pauses. Which actions should you take to resolve these performance concerns?

A. Optimize the a!queryEntity() calls by adding a!queryFilter() to reduce the result set
B. Increase the server's garbage collection frequency to reduce pauses
C. Use local variables instead of process variables to store query results in the interface
D. Implement database views to pre-aggregate data before querying from Appian

Answer: A, C, D

Explanation: Adding a!queryFilter() to a!queryEntity() reduces the number of rows retrieved, improving query and interface performance. Using local variables in the interface minimizes memory usage compared to process variables, which persist across the process. Implementing database views to pre-aggregate data reduces the data processed by Appian, improving load times. Increasing garbage collection frequency may worsen performance by causing more frequent pauses, so it is not a solution.

Question: 472

In a telecom application, you are designing a data model for customers, subscriptions, and plans. Each customer has multiple subscriptions, and each subscription is linked to one plan. The database must support queries for total subscriptions by plan. Which schema is best?

A. Two tables: Customers (customer_id, name, subscription_id, plan_id), Subscriptions (subscription_id, plan_id, start_date)
B. Three tables: Customers (customer_id, name), Subscriptions (subscription_id, customer_id, plan_id, start_date), Plans (plan_id, name)
C. Three tables: Customers (customer_id, name), Subscriptions (subscription_id, customer_id, start_date), Plans (plan_id, subscription_id, name)
D. Four tables: Customers (customer_id, name), Subscriptions (subscription_id, customer_id, start_date), Plans (plan_id, name), Subscription_Plans (subscription_id, plan_id)

Answer: B

Explanation: The schema with Customers, Subscriptions, and Plans tables models the one-to-many relationships (customer to subscriptions, plan to subscriptions) correctly. It supports querying subscriptions by plan via joins, ensuring normalization and efficiency. Other options introduce redundancy or incorrect relationships.

Question: 473

You are configuring security for an Appian application with a record type ProjectData. The record type includes a field budget that should only be editable by the ProjectManager group. Other groups, such as TeamMember, should view the field but not edit it. Which configuration achieves this?

A. Use restrictFields("budget", ["ProjectManager"], ["TeamMember"])
B. Set ProjectManager to Editor role for ProjectData
C. Use restrictFields("budget", ["ProjectManager"], [], ["TeamMember"])
D. Set TeamMember to Viewer role for ProjectData

Answer: C, D

Explanation: The restrictFields("budget", ["ProjectManager"], [], ["TeamMember"]) function allows ProjectManager to edit budget, denies editing to others, and permits TeamMember to view it. Setting TeamMember to Viewer role ensures they can view all fields without editing. The incorrect restrictFields syntax in option A omits the edit restriction, and setting ProjectManager to Editor is unnecessary.

Question: 474

In an insurance application, a Record Type named PolicyRecord is used to display policy details from a data store connected to a MySQL database. The record type uses data sync with a 20-minute refresh interval. A new requirement mandates that policy cancellations are reflected in the record list within 2 seconds. Which configuration meets this requirement?

A. Modify the record type to use a custom sync expression for cancellations
B. Create a process model to trigger a!refreshRecordData() for cancellations
C. Disable data sync and use a!queryRecordType() for real-time updates
D. Use a database trigger to notify Appian via a web API

Answer: A

Explanation: A custom sync expression prioritizes policy cancellations, meeting the 2-second requirement with minimal overhead. A process model for refreshing data is resource-intensive. Disabling data sync increases latency for all queries. A database trigger with a web API adds complexity and latency.

Question: 475

A financial application in Appian retrieves transaction data using a!queryEntity() with a complex join across three tables, each with 50,000 rows. The query takes 15 seconds to execute, impacting user experience. Which optimization techniques should you apply?

A. Add indexes on the join columns
B. Use a materialized view to store the joined data
C. Denormalize the tables to reduce joins
D. Increase the database connection pool size

Answer: A, B, C

Explanation: Indexes on join columns speed up data retrieval for the query. A materialized view precomputes the joined data, reducing query time for static datasets. Denormalizing tables reduces the need for joins, improving performance. Increasing the connection pool size does not optimize the query itself.

Question: 476

You are configuring a data store for an Appian application connecting to a MySQL table 'Tickets' (ticket_id, event_id, purchase_date, price). The CDT must support concurrent updates with minimal conflicts. Which configuration is best?

A. Configure a database trigger to manage conflicts and map the CDT without locking
B. Enable optimistic locking with @Version and use a!writeToDataStoreEntity() for updates
C. Use pessimistic locking in the CDT and configure a process model for updates

D. Set up auto-increment on ticket_id and disable locking

Answer: B

Explanation: Optimistic locking with the @Version annotation detects conflicts during concurrent updates, and a!writeToDataStoreEntity() supports efficient updates. Pessimistic locking is not supported, triggers are less integrated, and disabling locking risks data integrity.

Question: 477

In an Appian application for managing employee expenses, a custom plug-in is developed to integrate with an external payment gateway. The plug-in processes payment requests and returns transaction IDs. To ensure reliability, the plug-in must handle network timeouts and retry failed requests. Which configurations are necessary to implement this functionality?

A. Implement exponential backoff for retry logic
B. Use Appian's HTTP Smart Service for payment requests
C. Configure timeout settings in the plug-in's HTTP client
D. Log retry attempts for debugging

Answer: A, C, D

Explanation: Implementing exponential backoff for retry logic ensures that retries are spaced out to avoid overwhelming the external system. Configuring timeout settings in the plug-in's HTTP client prevents the plug-in from hanging on slow responses. Logging retry attempts aids in debugging and monitoring reliability. Using Appian's HTTP Smart Service is not appropriate, as the plug-in should encapsulate the payment logic for reusability and modularity.

Question: 478

You are optimizing a materialized view for an Appian application analyzing customer purchase patterns. The view aggregates data from the Purchase, Customer, and Product tables, grouping by CustomerID and ProductCategory. The view is refreshed every 12 hours, but queries are slow due to frequent filtering on ProductCategory. Which action will most effectively improve query performance?

A. Add a non-clustered index on ProductCategory
B. Use a stored procedure to rebuild the view
C. Increase the refresh interval to 24 hours
D. Use a!queryEntity() with pagination

Answer: A

Explanation: Adding a non-clustered index on ProductCategory will significantly improve query

performance, as it allows the database to quickly locate rows filtered by ProductCategory, which is frequently used in queries. A stored procedure to rebuild the view adds complexity without addressing query speed. Increasing the refresh interval does not improve query performance. Pagination in a!queryEntity() helps Appian performance but not the view's query speed.

Question: 479

You are designing a relational data model for a healthcare application in Appian that tracks patient visits, doctors, and medical procedures. Each patient can have multiple visits, each visit is associated with exactly one doctor, and multiple procedures can be performed during a visit. A procedure can be performed across multiple visits. The database must support efficient querying for a report showing all procedures performed by a doctor in a given month. Which database schema design best supports this requirement while adhering to third normal form (3NF)?

A. Three tables: Patients (patient_id, name), Visits (visit_id, patient_id, doctor_id, date), Procedures (procedure_id, visit_id, procedure_name)
B. Four tables: Patients (patient_id, name), Doctors (doctor_id, name), Visits (visit_id, patient_id, doctor_id, date), Visit_Procedures (visit_id, procedure_id, procedure_name)
C. Five tables: Patients (patient_id, name), Doctors (doctor_id, name), Visits (visit_id, patient_id, doctor_id, date), Procedures (procedure_id, name), Visit_Procedures (visit_id, procedure_id)
D. Two tables: Patients (patient_id, name, doctor_id, visit_id, date), Procedures (procedure_id, visit_id, procedure_name)

Answer: C

Explanation: To satisfy the requirements and adhere to 3NF, the schema must normalize relationships and avoid redundancy. The correct design requires separate tables for Patients, Doctors, Visits, Procedures, and a junction table (Visit_Procedures) to handle the many-to-many relationship between visits and procedures. This ensures that procedure names are stored only once in the Procedures table, and the Visit_Procedures table links visits to procedures efficiently. The schema supports the report query by allowing joins between Doctors, Visits, and Visit_Procedures to filter by date and doctor. Other options either fail to normalize the many-to-many relationship or introduce redundancy, violating 3NF.

Question: 480
In an Appian application for managing legal cases, a Smart Service is used to generate case summaries based on case data and related documents. The Smart Service must handle large datasets and ensure high availability. Which configurations ensure the Smart Service is robust and scalable?

A. Implement caching for frequently accessed data
B. Configure the Smart Service to run synchronously
C. Use connection pooling for database queries
D. Monitor performance metrics via the Appian Health Check

Answer: A, C, D

Explanation: Implementing caching for frequently accessed data reduces database load and improves

performance. Using connection pooling for database queries enhances scalability by reusing connections. Monitoring performance metrics via the Appian Health Check helps identify and address bottlenecks. Running the Smart Service synchronously can block the process, reducing scalability.

Question: 481

Your Appian application includes a database trigger that automatically updates a status column in a table when a record is modified. The trigger is defined as follows:

```
CREATE TRIGGER update_status
AFTER UPDATE ON orders
FOR EACH ROW
BEGIN
IF NEW.total_amount > 1000 THEN
SET NEW.status = 'High Value';
ELSE
SET NEW.status = 'Standard';
END IF;
END;
```

During testing, you notice that updates to the orders table via a Write to Data Store Entity smart service occasionally fail with a “trigger execution error.” What is the most likely cause of this issue?

A. The smart service does not have sufficient database permissions to execute the trigger.
B. The trigger attempts to modify the NEW.status column, which is not allowed in an AFTER UPDATE trigger.
C. The trigger syntax is incorrect due to a missing delimiter.
D. The data store entity does not include the status column in its CDT.

Answer: B

Explanation: In an AFTER UPDATE trigger, modifying the NEW record’s columns is not allowed, as the update has already occurred. The trigger attempts to set NEW.status, which causes an error. The smart service’s permissions, trigger syntax, and CDT configuration are not directly related to this issue, as the error stems from the trigger’s logic.

Question: 482

In an Appian application for managing clinical trials, a custom plug-in integrates with a lab management system to retrieve test results. The plug-in uses REST APIs and must handle authentication via JWT tokens. Which steps ensure secure implementation?

A. Store JWT tokens in a connected system object
B. Use HTTPS for API calls

C. Log JWT tokens for debugging
D. Implement token refresh logic

Answer: A, B, D

Explanation: Storing JWT tokens in a connected system object ensures secure credential management. Using HTTPS protects data in transit. Implementing token refresh logic ensures the plug-in remains functional when tokens expire. Logging JWT tokens is a security risk and should be avoided.

Question: 483

A healthcare application uses a Record Type named PatientVisit to display visit details from a data store backed by a MySQL database. The record type includes a calculated field that determines patient priority based on visit frequency. Users report slow performance when sorting the record list by this field. Which optimization improves sorting performance?

A. Cache the calculated field results in a separate CDT
B. Move the calculated field logic to a stored procedure
C. Create a database index on the fields used in the calculated field
D. Use a!queryRecordType() with pre-computed sorting

Answer: C

Explanation: A database index on the fields used in the calculated field optimizes sorting by enabling efficient data retrieval. A stored procedure adds complexity without leveraging Appian's query engine. Caching in a CDT risks stale data. Using a!queryRecordType() with pre-computed sorting does not address the root cause of slow database queries.

Question: 484

A retail application uses a data store to manage a table of customer orders, with a CDT named OrderDetails that includes fields for order ID, total, and status. The application experiences performance issues when querying orders due to a large number of nested CDTs. Which two actions optimize query performance?

A. Denormalize the CDT structure to reduce nested relationships
B. Create database indexes on frequently queried fields
C. Increase the database connection pool size in Appian
D. Use a!queryEntity() with batch processing for large queries

Answer: A, B

Explanation: Denormalizing the CDT structure simplifies queries by reducing joins, improving

performance. Creating database indexes on frequently queried fields speeds up data retrieval. Increasing the connection pool size does not address query complexity. Batch processing with a!queryEntity() helps with large datasets but does not optimize the data model.

Question: 485

Your organization is implementing an Appian application for managing customer feedback. A subprocess handles feedback analysis by calling an external sentiment analysis API. The subprocess must handle API rate limits and retry failed requests. Which configuration ensures reliable API integration?

A. Enable activity chaining for API calls
B. Store API responses in a process variable
C. Use a Timer Event to delay API calls
D. Implement retry logic with exponential backoff

Answer: D

Explanation: Implementing retry logic with exponential backoff ensures reliable API integration by spacing out retries to respect rate limits and handle transient failures. Activity chaining is unrelated to API reliability. A Timer Event delays calls but does not address rate limits or failures. Storing API responses in a process variable increases memory usage without improving reliability.

Question: 486

You are designing a database trigger for an Appian application managing inventory. The trigger must update a summary table with total stock whenever items are added or removed from the inventory table. Given the SQL below, which modification ensures accurate stock updates?

```
CREATE TRIGGER update_stock_summary
AFTER INSERT ON inventory
FOR EACH ROW
BEGIN
UPDATE stock_summary
SET total_stock = total_stock + NEW.quantity
WHERE warehouse_id = NEW.warehouse_id;
END;
```

A. Change AFTER INSERT to BEFORE INSERT
B. Add a trigger for DELETE operations
C. The trigger is correct as written
D. Replace NEW.quantity with OLD.quantity

Answer: B

Explanation: The trigger handles INSERT operations but does not account for DELETE operations, which would also affect total stock. Adding a DELETE trigger to subtract quantities ensures accurate updates. BEFORE INSERT is unnecessary as stock updates should occur after the insert commits. OLD.quantity is not available for INSERT triggers, and the trigger is incomplete without DELETE handling.

Question: 487

You are developing a process model that integrates with an external system via a web API. The process model uses a Call Integration smart service to invoke a POST request, passing a JSON payload constructed using an expression rule. During testing, the integration fails with a 401 Unauthorized error. The connected system is configured to use OAuth 2.0 Client Credentials Grant for authentication. Which two actions should you take to resolve the authentication issue?

A. Verify that the client ID and client secret are correctly configured in the connected system.
B. Switch the authentication method to Basic Authentication for simplicity.
C. Ensure the connected system has a valid access token endpoint URL.
D. Modify the expression rule to include authentication headers in the JSON payload.

Answer: A, C

Explanation: For OAuth 2.0 Client Credentials Grant, the connected system must be configured with a valid client ID, client secret, and access token endpoint URL to obtain an access token. Verifying these configurations ensures proper authentication. Including authentication headers in the JSON payload is incorrect, as Appian's connected system handles authentication automatically. Switching to Basic Authentication is not necessary and may not meet security requirements.

KILLEXAMS.COM

Killexams.com is a leading online platform specializing in high-quality certification exam preparation. Offering a robust suite of tools, including MCQs, practice tests, and advanced test engines, Killexams.com empowers candidates to excel in their certification exams. Discover the key features that make Killexams.com the go-to choice for exam success.

Exam Questions:

Killexams.com provides exam questions that are experienced in test centers. These questions are updated regularly to ensure they are up-to-date and relevant to the latest exam syllabus. By studying these questions, candidates can familiarize themselves with the content and format of the real exam.

Exam MCQs:

Killexams.com offers exam MCQs in PDF format. These questions contain a comprehensive collection of questions and answers that cover the exam topics. By using these MCQs, candidate can enhance their knowledge and improve their chances of success in the certification exam.

Practice Test:

Killexams.com provides practice test through their desktop test engine and online test engine. These practice tests simulate the real exam environment and help candidates assess their readiness for the actual exam. The practice test cover a wide range of questions and enable candidates to identify their strengths and weaknesses.

Guaranteed Success:

Killexams.com offers a success guarantee with the exam MCQs. Killexams claim that by using this materials, candidates will pass their exams on the first attempt or they will get refund for the purchase price. This guarantee provides assurance and confidence to individuals preparing for certification exam.

Updated Contents:

Killexams.com regularly updates its question bank of MCQs to ensure that they are current and reflect the latest changes in the exam syllabus. This helps candidates stay up-to-date with the exam content and increases their chances of success.