



Download Appian ACD301 Exam Dumps Free

Shared by Dawson on 17-06-2026

For More Free Questions and Preparation Resources

Check the Links on Last Page



Question 1

Question Type: MultipleChoice

You are asked to design a case management system for a client. In addition to storing some basic metadata about a case, one of the client's requirements is the ability for users to update a case. The client would like any user in their organization of 500 people to be able to make these updates. The users are all based in the company's headquarters, and there will be frequent cases where users are attempting to edit the same case. The client wants to ensure no information is lost when these edits occur and does not want the solution to burden their process administrators with any additional effort. Which data locking approach should you recommend?

Options:

- A- Allow edits without locking the case CDI.
- B- Use the database to implement low-level pessimistic locking.
- C- Add an @Version annotation to the case CDT to manage the locking.
- D- Design a process report and query to determine who opened the edit form first.

Answer:

C

Explanation:

Comprehensive and Detailed In-Depth Explanation:

The requirement involves a case management system where 500 users may simultaneously edit the same case, with a need to prevent data loss and minimize administrative overhead. Appian's data management and concurrency control strategies are critical here, especially when integrating with an underlying database.

Option C (Add an @Version annotation to the case CDT to manage the locking):

This is the recommended approach. In Appian, the @Version annotation on a Custom Data Type (CDT) enables optimistic locking, a lightweight concurrency control mechanism. When a user updates a case, Appian checks the version number of the CDT instance. If another user has modified it in the meantime, the update fails, prompting the user to refresh and reapply changes. This prevents data loss without requiring manual intervention by process administrators. Appian's Data Design Guide recommends @Version for scenarios with high concurrency (e.g., 500 users) and frequent edits, as it leverages the database's native versioning (e.g., in MySQL or PostgreSQL) and integrates seamlessly with Appian's process models. This aligns with the client's no-burden requirement.

Option A (Allow edits without locking the case CDI):

This is risky. Without locking, simultaneous edits could overwrite each other, leading to data loss--a direct violation of the client's requirement. Appian does not recommend this for collaborative environments.

Option B (Use the database to implement low-level pessimistic locking):

Pessimistic locking (e.g., using SELECT ... FOR UPDATE in MySQL) locks the record during the edit process, preventing other users from modifying it until the lock is released. While effective, it can lead to deadlocks or performance bottlenecks with 500 users, especially if edits are frequent. Additionally, managing this at the database level requires custom SQL and increases administrative effort (e.g., monitoring locks), which the client wants to avoid. Appian prefers higher-level solutions like @Version over low-level database locking.

Option D (Design a process report and query to determine who opened the edit form first):

This is impractical and inefficient. Building a custom report and query to track form opens adds complexity and administrative overhead. It doesn't inherently prevent data loss and relies on manual resolution, conflicting with the client's requirements.

The @Version annotation provides a robust, Appian-native solution that balances concurrency, data integrity, and ease of maintenance, making it the best fit.

Question 2

Question Type: MultipleChoice

Your application contains a process model that is scheduled to run daily at a certain time, which kicks off a user input task to a specified user on the 1st time zone for morning data collection. The time zone is set to the (default) pm!timezone. In this situation, what does the pm!timezone reflect?

Options:

- A- The time zone of the server where Appian is installed.
- B- The time zone of the user who most recently published the process model.
- C- The default time zone for the environment as specified in the Administration Console.
- D- The time zone of the user who is completing the input task.

Answer:

C

Explanation:

Comprehensive and Detailed In-Depth Explanation:

In Appian, the `pm!timezone` variable is a process variable automatically available in process models, reflecting the time zone context for scheduled or time-based operations. Understanding its behavior is critical for scheduling tasks accurately, especially in scenarios like this where a process runs daily and assigns a user input task.

Option C (The default time zone for the environment as specified in the Administration Console):

This is the correct answer. Per Appian's Process Model documentation, when a process model uses `pm!timezone` and no custom time zone is explicitly set, it defaults to the environment's time zone configured in the Administration Console (under System > Time Zone settings). For scheduled processes, such as one running "daily at a certain time," Appian uses this default time zone to determine when the process triggers. In this case, the task assignment occurs based on the schedule, and `pm!timezone` reflects the environment's setting, not the user's location.

Option A (The time zone of the server where Appian is installed): This is incorrect. While the server's time zone might influence underlying system operations, Appian abstracts this through the Administration Console's time zone setting. The `pm!timezone` variable aligns with the configured environment time zone, not the raw server setting.

Option B (The time zone of the user who most recently published the process model): This is irrelevant. Publishing a process model does not tie `pm!timezone` to the publisher's time zone. Appian's scheduling is system-driven, not user-driven in this context.

Option D (The time zone of the user who is completing the input task): This is also incorrect. While Appian can adjust task display times in the user interface to the assigned user's time zone (based on their profile settings), the `pm!timezone` in the process model reflects the environment's default time zone for scheduling purposes, not the assignee's.

For example, if the Administration Console is set to EST (Eastern Standard Time), the process will trigger daily at the specified time in EST, regardless of the assigned user's location. The "1st time zone" phrasing in the question appears to be a typo or miscommunication, but it doesn't change the fact that `pm!timezone` defaults to the environment setting.

Question 3

Question Type: MultipleChoice

A customer wants to integrate a CSV file once a day into their Appian application, sent every night at 1:00 AM. The file contains hundreds of thousands of items to be used daily by users as soon as their workday starts at 8:00 AM. Considering the high volume of data to manipulate and the nature of the operation, what is the best technical option to process the requirement?

Options:

- A- Use an Appian Process Model, initiated after every integration, to loop on each item and update it to the business requirements.
- B- Build a complex and optimized view (relevant indices, efficient joins, etc.), and use it every time a user needs to use the data.
- C- Create a set of stored procedures to handle the volume and the complexity of the expectations, and call it after each integration.
- D- Process what can be completed easily in a process model after each integration, and complete the most complex tasks using a set of stored procedures.

Answer:

C

Explanation:

Comprehensive and Detailed In-Depth Explanation:

As an Appian Lead Developer, handling a daily CSV integration with hundreds of thousands of items requires a solution that balances performance, scalability, and Appian's architectural strengths. The timing (1:00 AM integration, 8:00 AM availability) and data volume necessitate efficient processing and minimal runtime overhead. Let's evaluate each option based on Appian's official documentation and best practices:

A . Use an Appian Process Model, initiated after every integration, to loop on each item and update it to the business requirements:

This approach involves parsing the CSV in a process model and using a looping mechanism (e.g., a subprocess or script task with `fn!forEach`) to process each item. While Appian process models are excellent for orchestrating workflows, they are not optimized for high-volume data processing. Looping over hundreds of thousands of records would strain the process engine, leading to timeouts, memory issues, or slow execution---potentially missing the 8:00 AM deadline. Appian's documentation warns against using process models for bulk data operations, recommending database-level processing instead. This is not a viable solution.

B . Build a complex and optimized view (relevant indices, efficient joins, etc.), and use it every time a user needs to use the data:

This suggests loading the CSV into a table and creating an optimized database view (e.g., with indices and joins) for user queries via `a!queryEntity`. While this improves read performance for users at 8:00 AM, it doesn't address the integration process itself. The question focuses on processing the CSV ("manipulate" and "operation"), not just querying. Building a view assumes the data is already loaded and transformed, leaving the heavy lifting of integration unaddressed. This option is incomplete and misaligned with the requirement's focus on processing efficiency.

C . Create a set of stored procedures to handle the volume and the complexity of the expectations, and call it after each integration:

This is the best choice. Stored procedures, executed in the database, are designed for high-volume data manipulation (e.g., parsing CSV, transforming data, and applying business logic). In this scenario, you can configure an Appian process model to trigger at 1:00 AM (using a timer event) after the CSV is received (e.g., via FTP or Appian's File System utilities), then call a stored procedure via the "Execute Stored Procedure" smart service. The stored procedure can efficiently bulk-load the CSV (e.g., using SQL's BULK INSERT or equivalent), process the data, and update tables---all within the database's optimized environment. This ensures completion by 8:00 AM and aligns with Appian's recommendation to offload complex, large-scale data operations to the database layer, maintaining Appian as the orchestration layer.

D . Process what can be completed easily in a process model after each integration, and complete the most complex tasks using a set of stored procedures:

This hybrid approach splits the workload: simple tasks (e.g., validation) in a process model, and complex tasks (e.g., transformations) in stored procedures. While this leverages Appian's strengths (orchestration) and database efficiency, it adds unnecessary complexity. Managing two layers of processing increases maintenance overhead and risks partial failures (e.g., process model timeouts before stored procedures run). Appian's best practices favor a single, cohesive approach for bulk data integration, making this less efficient than a pure stored procedure solution (C).

Conclusion: Creating a set of stored procedures (C) is the best option. It leverages the database's native capabilities to handle the high volume and complexity of the CSV integration, ensuring fast, reliable processing between 1:00 AM and 8:00 AM. Appian orchestrates the trigger and integration (e.g., via a process model), while the stored procedure performs the heavy lifting---aligning with Appian's performance guidelines for large-scale data operations.

Appian Documentation: 'Execute Stored Procedure Smart Service' (Process Modeling > Smart Services).

Appian Lead Developer Certification: Data Integration Module (Handling Large Data Volumes).

Appian Best Practices: 'Performance Considerations for Data Integration' (Database vs. Process Model Processing).

Question 4

Question Type: MultipleChoice

An Appian application contains an integration used to send a JSON, called at the end of a form submission, returning the created code of the user request as the response. To be able to efficiently follow their case, the user needs to be informed of that code at the end of the process.

The JSON contains case fields (such as text, dates, and numeric fields) to a customer's API. What should be your two main considerations when building this integration?

Options:

- A- A process must be built to retrieve the API response afterwards so that the user experience is not impacted.
- B- The request must be a multi-part POST.
- C- The size limit of the body needs to be carefully followed to avoid an error.
- D- A dictionary that matches the expected request body must be manually constructed.

Answer:

C, D

Explanation:

Comprehensive and Detailed In-Depth Explanation:

As an Appian Lead Developer, building an integration to send JSON to a customer's API and return a code to the user involves balancing usability, performance, and reliability. The integration is triggered at form submission, and the user must see the response (case code) efficiently. The JSON includes standard fields (text, dates, numbers), and the focus is on primary considerations for the integration itself. Let's evaluate each option based on Appian's official documentation and best practices:

A . A process must be built to retrieve the API response afterwards so that the user experience is not impacted:

This suggests making the integration asynchronous by calling it in a process model (e.g., via a Start Process smart service) and retrieving the response later, avoiding delays in the UI. While this improves user experience for slow APIs (e.g., by showing a "Processing" message), it contradicts the requirement that the user is "informed of that code at the end of the process." Asynchronous processing would delay the code display, requiring additional steps (e.g., a follow-up task), which isn't efficient for this use case. Appian's default integration pattern (synchronous call in an Integration object) is suitable unless latency is a known issue, making this a secondary--not primary---consideration.

B . The request must be a multi-part POST:

A multi-part POST (e.g., multipart/form-data) is used for sending mixed content, like files and text, in a single request. Here, the payload is a JSON containing case fields (text, dates, numbers)---no files are mentioned. Appian's HTTP Connected System and Integration objects default to application/json for JSON payloads via a standard POST, which aligns with REST API norms. Forcing a multi-part POST adds unnecessary complexity and is incompatible with most

APIs expecting JSON. Appian documentation confirms this isn't required for JSON-only data, ruling it out as a primary consideration.

C . The size limit of the body needs to be carefully followed to avoid an error:

This is a primary consideration. Appian's Integration object has a payload size limit (approximately 10 MB, though exact limits depend on the environment and API), and exceeding it causes errors (e.g., 413 Payload Too Large). The JSON includes multiple case fields, and while "hundreds of thousands" isn't specified, large datasets could approach this limit. Additionally, the customer's API may impose its own size restrictions (common in REST APIs). Appian Lead Developer training emphasizes validating payload size during design---e.g., testing with maximum expected data---to prevent runtime failures. This ensures reliability and is critical for production success.

D . A dictionary that matches the expected request body must be manually constructed:

This is also a primary consideration. The integration sends a JSON payload to the customer's API, which expects a specific structure (e.g., { 'field1': 'text', 'field2': 'date' }). In Appian, the Integration object requires a dictionary (key-value pairs) to construct the JSON body, manually built to match the API's schema. Mismatches (e.g., wrong field names, types) cause errors (e.g., 400 Bad Request) or silent failures. Appian's documentation stresses defining the request body accurately---e.g., mapping form data to a CDT or dictionary---ensuring the API accepts the payload and returns the case code correctly. This is foundational to the integration's functionality.

Conclusion: The two primary considerations are C (size limit of the body) and D (constructing a matching dictionary). These ensure the integration works reliably (C) and meets the API's expectations (D), directly enabling the user to receive the case code at submission end. Size limits prevent technical failures, while the dictionary ensures data integrity---both are critical for a synchronous JSON POST in Appian. Option A could be relevant for performance but isn't primary given the requirement, and B is irrelevant to the scenario.

Appian Documentation: 'Integration Object' (Request Body Configuration and Size Limits).

Appian Lead Developer Certification: Integration Module (Building REST API Integrations).

Appian Best Practices: 'Designing Reliable Integrations' (Payload Validation and Error Handling).

Question 5

Question Type: MultipleChoice

You are required to create an integration from your Appian Cloud instance to an application hosted within a customer's self-managed environment.

The customer's IT team has provided you with a REST API endpoint to test with:

<https://internal.network/api/api/ping>.

Which recommendation should you make to progress this integration?

Options:

- A- Expose the API as a SOAP-based web service.
- B- Deploy the API/service into Appian Cloud.
- C- Add Appian Cloud's IP address ranges to the customer network's allowed IP listing.
- D- Set up a VPN tunnel.

Answer:

D

Explanation:

Comprehensive and Detailed In-Depth Explanation:

As an Appian Lead Developer, integrating an Appian Cloud instance with a customer's self-managed (on-premises) environment requires addressing network connectivity, security, and Appian's cloud architecture constraints. The provided endpoint (<https://internal.network/api/api/ping>) is a REST API on an internal network, inaccessible directly from Appian Cloud due to firewall restrictions and lack of public exposure. Let's evaluate each option:

A . Expose the API as a SOAP-based web service:

Converting the REST API to SOAP isn't a practical recommendation. The customer has provided a REST endpoint, and Appian fully supports REST integrations via Connected Systems and Integration objects. Changing the API to SOAP adds unnecessary complexity, development effort, and risks for the customer, with no benefit to Appian's integration capabilities. Appian's documentation emphasizes using the API's native format (REST here), making this irrelevant.

B . Deploy the API/service into Appian Cloud:

Deploying the customer's API into Appian Cloud is infeasible. Appian Cloud is a managed PaaS environment, not designed to host customer applications or APIs. The API resides in the customer's self-managed environment, and moving it would require significant architectural changes, violating security and operational boundaries. Appian's integration strategy focuses on connecting to external systems, not hosting them, ruling this out.

C . Add Appian Cloud's IP address ranges to the customer network's allowed IP listing:

This approach involves whitelisting Appian Cloud's IP ranges (available in Appian documentation) in the customer's firewall to allow direct HTTP/HTTPS requests. However, Appian Cloud's IPs are

dynamic and shared across tenants, making this unreliable for long-term integrations---changes in IP ranges could break connectivity. Appian's best practices discourage relying on IP whitelisting for cloud-to-on-premises integrations due to this limitation, favoring secure tunnels instead.

D . Set up a VPN tunnel:

This is the correct recommendation. A Virtual Private Network (VPN) tunnel establishes a secure, encrypted connection between Appian Cloud and the customer's self-managed network, allowing Appian to access the internal REST API (<https://internal.network/api/api/ping>). Appian supports VPNs for cloud-to-on-premises integrations, and this approach ensures reliability, security, and compliance with network policies. The customer's IT team can configure the VPN, and Appian's documentation recommends this for such scenarios, especially when dealing with internal endpoints.

Conclusion: Setting up a VPN tunnel (D) is the best recommendation. It enables secure, reliable connectivity from Appian Cloud to the customer's internal API, aligning with Appian's integration best practices for cloud-to-on-premises scenarios.

Appian Documentation: 'Integrating Appian Cloud with On-Premises Systems' (VPN and Network Configuration).

Appian Lead Developer Certification: Integration Module (Cloud-to-On-Premises Connectivity).

Appian Best Practices: 'Securing Integrations with Legacy Systems' (VPN Recommendations).

Question 6

Question Type: MultipleChoice

You have created a Web API in Appian with the following URL to call it:

https://exampleappiancloud.com/suite/webapi/user_management/users?username=john.smith.

Which is the correct syntax for referring to the username parameter?

Options:

- A- `httpRequest.queryParameters.users.username`
- B- `httpRequest.users.username`
- C- `httpRequest.formData.username`
- D- `httpRequest.queryParameters.username`

Answer:

D

Explanation:

Comprehensive and Detailed In-Depth Explanation:

In Appian, when creating a Web API, parameters passed in the URL (e.g., query parameters) are accessed within the Web API expression using the `httpRequest` object. The URL `https://exampleappiancloud.com/suite/webapi/user_management/users?username=john.smith` includes a query parameter `username` with the value `john.smith`. Appian's Web API documentation specifies how to handle such parameters in the expression rule associated with the Web API.

Option D (`httpRequest.queryParameters.username`):

This is the correct syntax. The `httpRequest.queryParameters` object contains all query parameters from the URL. Since `username` is a single query parameter, you access it directly as `httpRequest.queryParameters.username`. This returns the value `john.smith` as a text string, which can then be used in the Web API logic (e.g., to query a user record). Appian's expression language treats query parameters as key-value pairs under `queryParameters`, making this the standard approach.

Option A (`httpRequest.queryParameters.users.username`):

This is incorrect. The `users` part suggests a nested structure (e.g., `users` as a parameter containing a `username` subfield), which does not match the URL. The URL only defines `username` as a top-level query parameter, not a nested object.

Option B (`httpRequest.users.username`):

This is invalid. The `httpRequest` object does not have a direct `users` property. Query parameters are accessed via `queryParameters`, and there's no indication of a `users` object in the URL or Appian's Web API model.

Option C (`httpRequest.formData.username`):

This is incorrect. The `httpRequest.formData` object is used for parameters passed in the body of a POST or PUT request (e.g., form submissions), not for query parameters in a GET request URL. Since the `username` is part of the query string (`?username=john.smith`), `formData` does not apply.

The correct syntax leverages Appian's standard handling of query parameters, ensuring the Web API can process the `username` value effectively.

Question 7

Question Type: MultipleChoice

What are two advantages of having High Availability (HA) for Appian Cloud applications?

Options:

A- An Appian Cloud HA instance is composed of multiple active nodes running in different availability zones in different regions.

B- Data and transactions are continuously replicated across the active nodes to achieve redundancy and avoid single points of failure.

C- A typical Appian Cloud HA instance is composed of two active nodes.

D- In the event of a system failure, your Appian instance will be restored and available to your users in less than 15 minutes, having lost no more than the last 1 minute worth of data.

D- In the event of a system failure, your Appian instance will be restored and available to your users in less than 15 minutes, having lost no more than the last 1 minute worth of data. This is an advantage of having HA, as it guarantees a high level of service availability and reliability for your Appian instance. If one of the nodes fails or becomes unavailable, the other node will take over and continue to serve requests without any noticeable downtime or data loss for your users. The other options are incorrect for the following reasons:

A . An Appian Cloud HA instance is composed of multiple active nodes running in different availability zones in different regions. This is not an advantage of having HA, but rather a description of how HA works in Appian Cloud. An Appian Cloud HA instance consists of two active nodes running in different availability zones within the same region, not different regions.

C . A typical Appian Cloud HA instance is composed of two active nodes. This is not an advantage of having HA, but rather a description of how HA works in Appian Cloud. A typical Appian Cloud HA instance consists of two active nodes running in different availability zones within the same region, but this does not necessarily provide any benefit over having one active node. Verified Reference:Appian Documentation, section "High Availability".

Answer:

B, D, D

Explanation:

Comprehensive and Detailed In-Depth Explanation:

High Availability (HA) in Appian Cloud is designed to ensure that applications remain operational and data integrity is maintained even in the face of hardware failures, network issues, or other disruptions. Appian's Cloud Architecture and HA documentation outline the benefits, focusing on redundancy, minimal downtime, and data protection. The question asks for two advantages, and the options must align with these core principles.

Option B (Data and transactions are continuously replicated across the active nodes to achieve redundancy and avoid single points of failure):

This is a key advantage of HA. Appian Cloud HA instances use multiple active nodes to replicate data and transactions in real-time across the cluster. This redundancy ensures that if one node fails, others can take over without data loss, eliminating single points of failure. This is a fundamental feature of Appian's HA setup, leveraging distributed architecture to enhance reliability, as detailed in the Appian Cloud High Availability Guide.

Option D (In the event of a system failure, your Appian instance will be restored and available to your users in less than 15 minutes, having lost no more than the last 1 minute worth of data):

This is another significant advantage. Appian Cloud HA is engineered to provide rapid recovery and minimal data loss. The Service Level Agreement (SLA) and HA documentation specify that in the case of a failure, the system failover is designed to complete within a short timeframe (typically under 15 minutes), with data loss limited to the last minute due to synchronous replication. This ensures business continuity and meets stringent uptime and data integrity requirements.

Option A (An Appian Cloud HA instance is composed of multiple active nodes running in different availability zones in different regions):

This is a description of the HA architecture rather than an advantage. While running nodes across different availability zones and regions enhances fault tolerance, the benefit is the resulting redundancy and availability, which are captured in Options B and D. This option is more about implementation than a direct user or operational advantage.

Option C (A typical Appian Cloud HA instance is composed of two active nodes):

This is a factual statement about the architecture but not an advantage. The number of nodes (typically two or more, depending on configuration) is a design detail, not a benefit. The advantage lies in what this setup enables (e.g., redundancy and quick recovery), as covered by B and D.

The two advantages---continuous replication for redundancy (B) and fast recovery with minimal data loss (D)---reflect the primary value propositions of Appian Cloud HA, ensuring both operational resilience and data integrity for users.

The two advantages of having High Availability (HA) for Appian Cloud applications are:

B . Data and transactions are continuously replicated across the active nodes to achieve redundancy and avoid single points of failure. This is an advantage of having HA, as it ensures that there is always a backup copy of data and transactions in case one of the nodes fails or becomes unavailable. This also improves data integrity and consistency across the nodes, as any changes made to one node are automatically propagated to the other node.

Question 8

Question Type: MultipleChoice

You are required to configure a connection so that Jira can inform Appian when specific tickets change (using a webhook). Which three required steps will allow you to connect both systems?

Options:

- A- Create a Web API object and set up the correct security.
- B- Configure the connection in Jira specifying the URL and credentials.
- C- Create a new API Key and associate a service account.
- D- Give the service account system administrator privileges.
- E- Create an integration object from Appian to Jira to periodically check the ticket status.

Answer:

A, B, C

Explanation:

Comprehensive and Detailed In-Depth Explanation:

Configuring a webhook connection from Jira to Appian requires setting up a mechanism for Jira to push ticket change notifications to Appian in real-time. This involves creating an endpoint in Appian to receive the webhook and configuring Jira to send the data. Appian's Integration Best Practices and Web API documentation provide the framework for this process.

Option A (Create a Web API object and set up the correct security):

This is a required step. In Appian, a Web API object serves as the endpoint to receive incoming webhook requests from Jira. You must define the API structure (e.g., HTTP method, input parameters) and configure security (e.g., basic authentication, API key, or OAuth) to validate incoming requests. Appian recommends using a service account with appropriate permissions to ensure secure access, aligning with the need for a controlled webhook receiver.

Option B (Configure the connection in Jira specifying the URL and credentials):

This is essential. In Jira, you need to set up a webhook by providing the Appian Web API's URL (e.g., `https://suite/webapi/<web-api-name>`) and the credentials or authentication method (e.g., API key or basic auth) that match the security setup in Appian. This ensures Jira can successfully send ticket change events to Appian.

Option C (Create a new API Key and associate a service account):

This is necessary for secure authentication. Appian recommends using an API key tied to a service account for webhook integrations. The service account should have permissions to process the incoming data (e.g., write to a process or data store) but not excessive privileges. This step complements the Web API security setup and Jira configuration.

Option D (Give the service account system administrator privileges):

This is unnecessary and insecure. System administrator privileges grant broad access, which is overkill for a webhook integration. Appian's security best practices advocate for least-privilege principles, limiting the service account to the specific objects or actions needed (e.g., executing the Web API).

Option E (Create an integration object from Appian to Jira to periodically check the ticket status):

This is incorrect for a webhook scenario. Webhooks are push-based, where Jira notifies Appian of changes. Creating an integration object for periodic polling (pull-based) is a different approach and not required for the stated requirement of Jira informing Appian via webhook.

These three steps (A, B, C) establish a secure, functional webhook connection without introducing unnecessary complexity or security risks.

The three required steps that will allow you to connect both systems are:

A . Create a Web API object and set up the correct security. This will allow you to define an endpoint in Appian that can receive requests from Jira via webhook. You will also need to configure the security settings for the Web API object, such as authentication method, allowed origins, and access control.

B . Configure the connection in Jira specifying the URL and credentials. This will allow you to set up a webhook in Jira that can send requests to Appian when specific tickets change. You will need to specify the URL of the Web API object in Appian, as well as any credentials required for authentication.

C . Create a new API Key and associate a service account. This will allow you to generate a unique token that can be used for authentication between Jira and Appian. You will also need to create a service account in Appian that has permissions to access or update data related to Jira tickets.

The other options are incorrect for the following reasons:

D . Give the service account system administrator privileges. This is not required and could pose a security risk, as giving system administrator privileges to a service account could allow it to perform actions that are not related to Jira tickets, such as modifying system settings or accessing sensitive data.

E . Create an integration object from Appian to Jira to periodically check the ticket status. This is not required and could cause unnecessary overhead, as creating an integration object from Appian to Jira would involve polling Jira for ticket status changes, which could consume more resources than using webhook notifications. Verified Reference: Appian Documentation, section "Web API" and "API Keys".

Question 9

Question Type: MultipleChoice

You are designing a process that is anticipated to be executed multiple times a day. This process retrieves data from an external system and then calls various utility processes as needed. The main process will not use the results of the utility processes, and there are no user forms anywhere.

Which design choice should be used to start the utility processes and minimize the load on the execution engines?

Options:

- A- Use the Start Process Smart Service to start the utility processes.
- B- Start the utility processes via a subprocess synchronously.
- C- Use Process Messaging to start the utility process.
- D- Start the utility processes via a subprocess asynchronously.

Answer:

D

Explanation:

Comprehensive and Detailed In-Depth Explanation:

As an Appian Lead Developer, designing a process that executes frequently (multiple times a day) and calls utility processes without using their results requires optimizing performance and minimizing load on Appian's execution engines. The absence of user forms indicates a backend process, so user experience isn't a concern---only engine efficiency matters. Let's evaluate each option:

A . Use the Start Process Smart Service to start the utility processes:

The Start Process Smart Service launches a new process instance independently, creating a separate process in the Work Queue. While functional, it increases engine load because each utility process runs as a distinct instance, consuming engine resources and potentially clogging the Java Work Queue, especially with frequent executions. Appian's performance guidelines discourage unnecessary separate process instances for utility tasks, favoring integrated subprocesses, making this less optimal.

B . Start the utility processes via a subprocess synchronously:

Synchronous subprocesses (e.g., `a!startProcess` with `isAsync: false`) execute within the main

process flow, blocking until completion. For utility processes not used by the main process, this creates unnecessary delays, increasing execution time and engine load. With frequent daily executions, synchronous subprocesses could strain engines, especially if utility processes are slow or numerous. Appian's documentation recommends asynchronous execution for non-dependent, non-blocking tasks, ruling this out.

C . Use Process Messaging to start the utility process:

Process Messaging (e.g., `sendMessage()` in Appian) is used for inter-process communication, not for starting processes. It's designed to pass data between running processes, not initiate new ones. Attempting to use it for starting utility processes would require additional setup (e.g., a listening process) and isn't a standard or efficient method. Appian's messaging features are for coordination, not process initiation, making this inappropriate.

D . Start the utility processes via a subprocess asynchronously:

This is the best choice. Asynchronous subprocesses (e.g., `a!startProcess` with `isAsync: true`) execute independently of the main process, offloading work to the engine without blocking or delaying the parent process. Since the main process doesn't use the utility process results and there are no user forms, asynchronous execution minimizes engine load by distributing tasks across time, reducing Work Queue pressure during frequent executions. Appian's performance best practices recommend asynchronous subprocesses for non-dependent, utility tasks to optimize engine utilization, making this ideal for minimizing load.

Conclusion: Starting the utility processes via a subprocess asynchronously (D) minimizes engine load by allowing independent execution without blocking the main process, aligning with Appian's performance optimization strategies for frequent, backend processes.

Appian Documentation: 'Process Model Performance' (Synchronous vs. Asynchronous Subprocesses).

Appian Lead Developer Certification: Process Design Module (Optimizing Engine Load).

Appian Best Practices: 'Designing Efficient Utility Processes' (Asynchronous Execution).

Question 10

Question Type: MultipleChoice

You are running an inspection as part of the first deployment process from TEST to PROD. You receive a notice that one of your objects will not deploy because it is dependent on an object from an application owned by a separate team.

What should be your next step?

Options:

- A- Create your own object with the same code base, replace the dependent object in the application, and deploy to PROD.
- B- Halt the production deployment and contact the other team for guidance on promoting the object to PROD.
- C- Check the dependencies of the necessary object. Deploy to PROD if there are few dependencies and it is low risk.
- D- Push a functionally viable package to PROD without the dependencies, and plan the rest of the deployment accordingly with the other team's constraints.

Answer:

B

Explanation:

Comprehensive and Detailed In-Depth Explanation:

As an Appian Lead Developer, managing a deployment from TEST to PROD requires careful handling of dependencies, especially when objects from another team's application are involved. The scenario describes a dependency issue during deployment, signaling a need for collaboration and governance. Let's evaluate each option:

A . Create your own object with the same code base, replace the dependent object in the application, and deploy to PROD:

This approach involves duplicating the object, which introduces redundancy, maintenance risks, and potential version control issues. It violates Appian's governance principles, as objects should be owned and managed by their respective teams to ensure consistency and avoid conflicts. Appian's deployment best practices discourage duplicating objects unless absolutely necessary, making this an unsustainable and risky solution.

B . Halt the production deployment and contact the other team for guidance on promoting the object to PROD:

This is the correct step. When an object from another application (owned by a separate team) is a dependency, Appian's deployment process requires coordination to ensure both applications' objects are deployed in sync. Halting the deployment prevents partial deployments that could break functionality, and contacting the other team aligns with Appian's collaboration and governance guidelines. The other team can provide the necessary object version, adjust their deployment timeline, or resolve the dependency, ensuring a stable PROD environment.

C . Check the dependencies of the necessary object. Deploy to PROD if there are few dependencies and it is low risk:

This approach risks deploying an incomplete or unstable application if the dependency isn't fully

resolved. Even with "few dependencies" and "low risk," deploying without the other team's object could lead to runtime errors or broken functionality in PROD. Appian's documentation emphasizes thorough dependency management during deployment, requiring all objects (including those from other applications) to be promoted together, making this risky and not recommended.

D . Push a functionally viable package to PROD without the dependencies, and plan the rest of the deployment accordingly with the other team's constraints:

Deploying without dependencies creates an incomplete solution, potentially leaving the application non-functional or unstable in PROD. Appian's deployment process ensures all dependencies are included to maintain application integrity, and partial deployments are discouraged unless explicitly planned (e.g., phased rollouts). This option delays resolution and increases risk, contradicting Appian's best practices for Production stability.

Conclusion: Halting the production deployment and contacting the other team for guidance (B) is the next step. It ensures proper collaboration, aligns with Appian's governance model, and prevents deployment errors, providing a safe and effective resolution.

Appian Documentation: 'Deployment Best Practices' (Managing Dependencies Across Applications).

Appian Lead Developer Certification: Application Management Module (Cross-Team Collaboration).

Appian Best Practices: 'Handling Production Deployments' (Dependency Resolution).

Question 11

Question Type: MultipleChoice

You are planning a strategy around data volume testing for an Appian application that queries and writes to a MySQL database. You have administrator access to the Appian application and to the database. What are two key considerations when designing a data volume testing strategy?

Options:

- A- Data from previous tests needs to remain in the testing environment prior to loading prepopulated data.
- B- Large datasets must be loaded via Appian processes.
- C- The amount of data that needs to be populated should be determined by the project sponsor and the stakeholders based on their estimation.
- D- Testing with the correct amount of data should be in the definition of done as part of each

sprint.

E- Data model changes must wait until towards the end of the project.

Answer:

C, D

Explanation:

Comprehensive and Detailed In-Depth Explanation:

Data volume testing ensures an Appian application performs efficiently under realistic data loads, especially when interacting with external databases like MySQL. As an Appian Lead Developer with administrative access, the focus is on scalability, performance, and iterative validation. The two key considerations are:

Option C (The amount of data that needs to be populated should be determined by the project sponsor and the stakeholders based on their estimation):

Determining the appropriate data volume is critical to simulate real-world usage. Appian's Performance Testing Best Practices recommend collaborating with stakeholders (e.g., project sponsors, business analysts) to define expected data sizes based on production scenarios. This ensures the test reflects actual requirements---like peak transaction volumes or record counts---rather than arbitrary guesses. For example, if the application will handle 1 million records in production, stakeholders must specify this to guide test data preparation.

Option D (Testing with the correct amount of data should be in the definition of done as part of each sprint):

Appian's Agile Development Guide emphasizes incorporating performance testing (including data volume) into the Definition of Done (DoD) for each sprint. This ensures that features are validated under realistic conditions iteratively, preventing late-stage performance issues. With admin access, you can query/write to MySQL and assess query performance or write latency with the specified data volume, aligning with Appian's recommendation to "test early and often."

Option A (Data from previous tests needs to remain in the testing environment prior to loading prepopulated data): This is impractical and risky. Retaining old test data can skew results, introduce inconsistencies, or violate data integrity (e.g., duplicate keys in MySQL). Best practices advocate for a clean, controlled environment with fresh, prepopulated data per test cycle.

Option B (Large datasets must be loaded via Appian processes): While Appian processes can load data, this is not a requirement. With database admin access, you can use SQL scripts or tools like MySQL Workbench for faster, more efficient data population, bypassing Appian process overhead. Appian documentation notes this as a preferred method for large datasets.

Option E (Data model changes must wait until towards the end of the project): Delaying data model changes contradicts Agile principles and Appian's iterative design approach. Changes

should occur as needed throughout development to adapt to testing insights, not be deferred.

Question 12

Question Type: MultipleChoice

You are just starting with a new team that has been working together on an application for months. They ask you to review some of their views that have been degrading in performance. The views are highly complex with hundreds of lines of SQL. What is the first step in troubleshooting the degradation?

Options:

- A- Go through the entire database structure to obtain an overview, ensure you understand the business needs, and then normalize the tables to optimize performance.
- B- Run an explain statement on the views, identify critical areas of improvement that can be remediated without business knowledge.
- C- Go through all of the tables one by one to identify which of the grouped by, ordered by, or joined keys are currently indexed.
- D- Browse through the tables, note any tables that contain a large volume of null values, and work with your team to plan for table restructure.

Answer:

B

Explanation:

Comprehensive and Detailed In-Depth Explanation:

Troubleshooting performance degradation in complex SQL views within an Appian application requires a systematic approach. The views, described as having hundreds of lines of SQL, suggest potential issues with query execution, indexing, or join efficiency. As a new team member, the first step should focus on quickly identifying the root cause without overhauling the system prematurely. Appian's Performance Troubleshooting Guide and database optimization best practices provide the framework for this process.

Option B (Run an explain statement on the views, identify critical areas of improvement that can be remediated without business knowledge):

This is the recommended first step. Running an EXPLAIN statement (or equivalent, such as EXPLAIN PLAN in some databases) analyzes the query execution plan, revealing details like full

table scans, missing indices, or inefficient joins. This technical analysis can identify immediate optimization opportunities (e.g., adding indices or rewriting subqueries) without requiring business input, allowing you to address low-hanging fruit quickly. Appian encourages using database tools to diagnose performance issues before involving stakeholders, making this a practical starting point as you familiarize yourself with the application.

Option A (Go through the entire database structure to obtain an overview, ensure you understand the business needs, and then normalize the tables to optimize performance):

This is too broad and time-consuming as a first step. Understanding business needs and normalizing tables are valuable but require collaboration with the team and stakeholders, delaying action. It's better suited for a later phase after initial technical analysis.

Option C (Go through all of the tables one by one to identify which of the grouped by, ordered by, or joined keys are currently indexed):

Manually checking indices is useful but inefficient without first knowing which queries are problematic. The EXPLAIN statement provides targeted insights into index usage, making it a more direct initial step than a manual table-by-table review.

Option D (Browse through the tables, note any tables that contain a large volume of null values, and work with your team to plan for table restructure):

Identifying null values and planning restructures is a long-term optimization strategy, not a first step. It requires team input and may not address the immediate performance degradation, which is better tackled with query-level diagnostics.

Starting with an EXPLAIN statement allows you to gather data-driven insights, align with Appian's performance troubleshooting methodology, and proceed with informed optimizations.



To Get Premium Files for ACD301 Visit

<https://www.p2pexams.com/products/acd301>

For More Free Questions Visit

<https://www.p2pexams.com/appian/pdf/acd301>

20%
DISCOUNT

P2P
exams