

選擇ACD-301考試資訊 -不用再擔心Appian Certified Lead Developer考試



ACD300

Appian Certified Lead Developer



從Google Drive中免費下載最新的PDFExamDumps ACD-301 PDF版考試題庫：https://drive.google.com/open?id=1ll_uBPKG0k8MKYEP1YIMsV1oGBauHbF9

在如今競爭激烈的IT行業中，通過了Appian ACD-301 認證考試是有很多好處的。因為有了Appian ACD-301 認證證書就可以提高收入。拿到了Appian ACD-301 認證證書的人往往要比沒有證書的同行工資高很多。可是Appian ACD-301 認證考試不是很容易通過的，所以PDFExamDumps是一個可以幫助你增長收入的網站。

我們PDFExamDumps全面提供Appian的ACD-301考試認證資料，為你提示成功。我們的培訓資料是由專家帶來的最新的研究材料，你總是得到最新的研究材料，保證你的成功會與我們PDFExamDumps同在，我們幫助你，你肯定從我們這裏得到最詳細最準確的考題及答案，我們培訓工具定期更新，不斷變化的考試目標。其實成功並不遠，你順著PDFExamDumps往下走，就一定能走向你專屬的成功之路。

>> ACD-301考試資訊 <<

ACD-301熱門證照 - ACD-301考試內容

難道你不想在你的工作生涯中做出一番輝煌的成績嗎？肯定希望那樣吧。那麼，你就有必要時常提升自己了。在Appian行業工作的你應該怎樣提升自己的水準呢？其實參加IT認證考試獲得認證資格是一個好方法。Appian的認證考試資格是很重要的資格，因此參加ACD-301考試的人變得越來越多了。

最新的 Appian Certification Program ACD-301 免費考試真題 (Q37-Q42):

問題 #37

While working on an application, you have identified oddities and breaks in some of your components. How can you guarantee that this mistake does not happen again in the future?

- A. Provide Appian developers with the "Designer" permissions role within Appian. Ensure that they have only basic user rights and assign them the permissions to administer their application.
- B. Ensure that the application administrator group only has designers from that application's team.
- **C. Create a best practice that enforces a peer review of the deletion of any components within the application.**
- D. Design and communicate a best practice that dictates designers only work within the confines of their own application.

答案：C

解題說明：

Comprehensive and Detailed In-Depth Explanation:

As an Appian Lead Developer, preventing recurring "oddities and breaks" in application components requires addressing root causes-likely tied to human error, lack of oversight, or uncontrolled changes-while leveraging Appian's governance and collaboration features. The question implies a past mistake (e.g., accidental deletions or modifications) and seeks a proactive, sustainable solution. Let's evaluate each option based on Appian's official documentation and best practices:

A. Design and communicate a best practice that dictates designers only work within the confines of their own application:

This suggests restricting designers to their assigned applications via a policy. While Appian supports application-level security (e.g., Designer role scoped to specific applications), this approach relies on voluntary compliance rather than enforcement. It doesn't directly address "oddities and breaks"-e.g., a designer could still mistakenly alter components within their own application. Appian's documentation emphasizes technical controls and process rigor over broad guidelines, making this insufficient as a guarantee.

B . Ensure that the application administrator group only has designers from that application's team:

This involves configuring security so only team-specific designers have Administrator rights to the application (via Appian's Security settings). While this limits external interference, it doesn't prevent internal mistakes (e.g., a team designer deleting a critical component). Appian's security model already restricts access by default, and the issue isn't about unauthorized access but rather component integrity. This step is a hygiene factor, not a direct solution to the problem, and fails to "guarantee" prevention.

C . Create a best practice that enforces a peer review of the deletion of any components within the application:

This is the best choice. A peer review process for deletions (e.g., process models, interfaces, or records) introduces a checkpoint to catch errors before they impact the application. In Appian, deletions are permanent and can cascade (e.g., breaking dependencies), aligning with the "oddities and breaks" described. While Appian doesn't natively enforce peer reviews, this can be implemented via team workflows-e.g., using Appian's collaboration tools (like Comments or Tasks) or integrating with version control practices during deployment. Appian Lead Developer training emphasizes change management and peer validation to maintain application stability, making this a robust, preventive measure that directly addresses the root cause.

D . Provide Appian developers with the "Designer" permissions role within Appian. Ensure that they have only basic user rights and assign them the permissions to administer their application:

This option is confusingly worded but seems to suggest granting Designer system role permissions (a high-level privilege) while limiting developers to Viewer rights system-wide, with Administrator rights only for their application. In Appian, the "Designer" system role grants broad platform access (e.g., creating applications), which contradicts "basic user rights" (Viewer role).

Regardless, adjusting permissions doesn't prevent mistakes-it only controls who can make them. The issue isn't about access but about error prevention, so this option misses the mark and is impractical due to its contradictory setup.

Conclusion: Creating a best practice that enforces a peer review of the deletion of any components (C) is the strongest solution. It directly mitigates the risk of "oddities and breaks" by adding oversight to destructive actions, leveraging team collaboration, and aligning with Appian's recommended governance practices. Implementation could involve documenting the process, training the team, and using Appian's monitoring tools (e.g., Application Properties history) to track changes-ensuring mistakes are caught before deployment. This provides the closest guarantee to preventing recurrence.

Appian Documentation: "Application Security and Governance" (Change Management Best Practices).

Appian Lead Developer Certification: Application Design Module (Preventing Errors through Process).

Appian Best Practices: "Team Collaboration in Appian Development" (Peer Review Recommendations).

問題 #38

You are developing a case management application to manage support cases for a large set of sites. One of the tabs in this application's site is a record grid of cases, along with information about the site corresponding to that case. Users must be able to filter cases by priority level and status.

You decide to create a view as the source of your entity-backed record, which joins the separate case/site tables (as depicted in the following image).

Which three columns should be indexed?

- A. modified_date
- B. priority
- C. site_id
- D. name
- E. status
- F. case_id

答案: B,C,E

解題說明:

Indexing columns can improve the performance of queries that use those columns in filters, joins, or order by clauses. In this case, the columns that should be indexed are site_id, status, and priority, because they are used for filtering or joining the tables. Site_id is used to join the case and site tables, so indexing it will speed up the join operation. Status and priority are used to filter the cases by the user's input, so indexing them will reduce the number of rows that need to be scanned. Name, modified_date, and case_id do not need to be indexed, because they are not used for filtering or joining. Name and modified_date are only used for displaying information in the record grid, and case_id is only used as a unique identifier for each record. Verified Appian Records Tutorial, Appian Best Practices As an Appian Lead Developer, optimizing a database view for an entity-backed record grid requires indexing columns frequently used in queries, particularly for filtering and joining. The scenario involves a record grid displaying cases with site information, filtered by "priority level" and "status," and joined via the site_id foreign key. The image shows two tables (site and case) with a relationship via site_id. Let's evaluate each column based on Appian's performance best practices and query

patterns:

A . site_id: This is a primary key in the site table and a foreign key in the case table, used for joining the tables in the view. Indexing site_id in the case table (and ensuring it's indexed in site as a PK) optimizes JOIN operations, reducing query execution time for the record grid. Appian's documentation recommends indexing foreign keys in large datasets to improve query performance, especially for entity-backed records. This is critical for the join and must be included.

B . status: Users filter cases by "status" (a varchar column in the case table). Indexing status speeds up filtering queries (e.g., WHERE status = 'Open') in the record grid, particularly with large datasets. Appian emphasizes indexing columns used in WHERE clauses or filters to enhance performance, making this a key column for optimization. Since status is a common filter, it's essential.

C . name: This is a varchar column in the site table, likely used for display (e.g., site name in the grid). However, the scenario doesn't mention filtering or sorting by name, and it's not part of the join or required filters. Indexing name could improve searches if used, but it's not a priority given the focus on priority and status filters. Appian advises indexing only frequently queried or filtered columns to avoid unnecessary overhead, so this isn't necessary here.

D . modified_date: This is a date column in the case table, tracking when cases were last updated. While useful for sorting or historical queries, the scenario doesn't specify filtering or sorting by modified_date in the record grid. Indexing it could help if used, but it's not critical for the current requirements. Appian's performance guidelines prioritize indexing columns in active filters, making this lower priority than site_id, status, and priority.

E . priority: Users filter cases by "priority level" (a varchar column in the case table). Indexing priority optimizes filtering queries (e.g., WHERE priority = 'High') in the record grid, similar to status. Appian's documentation highlights indexing columns used in WHERE clauses for entity-backed records, especially with large datasets. Since priority is a specified filter, it's essential to include.

F . case_id: This is the primary key in the case table, already indexed by default (as PKs are automatically indexed in most databases). Indexing it again is redundant and unnecessary, as Appian's Data Store configuration relies on PKs for unique identification but doesn't require additional indexing for performance in this context. The focus is on join and filter columns, not the PK itself.

Conclusion: The three columns to index are A (site_id), B (status), and E (priority). These optimize the JOIN (site_id) and filter performance (status, priority) for the record grid, aligning with Appian's recommendations for entity-backed records and large datasets. Indexing these columns ensures efficient querying for user filters, critical for the application's performance.

Appian Documentation: "Performance Best Practices for Data Stores" (Indexing Strategies).

Appian Lead Developer Certification: Data Management Module (Optimizing Entity-Backed Records).

Appian Best Practices: "Working with Large Data Volumes" (Indexing for Query Performance).

問題 #39

The business database for a large, complex Appian application is to undergo a migration between database technologies, as well as interface and process changes. The project manager asks you to recommend a test strategy. Given the changes, which two items should be included in the test strategy?

- A. Internationalization testing of the Appian platform
- B. Tests for each of the interfaces and process changes
- C. A regression test of all existing system functionality
- D. Tests that ensure users can still successfully log into the platform
- E. Penetration testing of the Appian platform

答案: B,C

解題說明:

Comprehensive and Detailed In-Depth Explanation:

As an Appian Lead Developer, recommending a test strategy for a large, complex application undergoing a database migration (e.g., from Oracle to PostgreSQL) and interface/process changes requires focusing on ensuring system stability, functionality, and the specific updates. The strategy must address risks tied to the scope-database technology shift, interface modifications, and process updates-while aligning with Appian's testing best practices. Let's evaluate each option:

A . Internationalization testing of the Appian platform:

Internationalization testing verifies that the application supports multiple languages, locales, and formats (e.g., date formats). While valuable for global applications, the scenario doesn't indicate a change in localization requirements tied to the database migration, interfaces, or processes. Appian's platform handles internationalization natively (e.g., via locale settings), and this isn't impacted by database technology or UI/process changes unless explicitly stated. This is out of scope for the given context and not a priority.

B . A regression test of all existing system functionality:

This is a critical inclusion. A database migration between technologies can affect data integrity, queries (e.g., a!queryEntity), and performance due to differences in SQL dialects, indexing, or drivers. Regression testing ensures that all existing functionality-records, reports, processes, and integrations-works as expected post-migration. Appian Lead Developer documentation mandates regression testing for significant infrastructure changes like this, as unmapped edge cases (e.g., datatype mismatches) could break the application. Given the "large, complex" nature, full-system validation is essential to catch unintended impacts.

C . Penetration testing of the Appian platform:

Penetration testing assesses security vulnerabilities (e.g., injection attacks). While security is important, the changes described- database migration, interface, and process updates- don't inherently alter Appian's security model (e.g., authentication, encryption), which is managed at the platform level. Appian's cloud or on-premise security isn't directly tied to database technology unless new vulnerabilities are introduced (not indicated here). This is a periodic concern, not specific to this migration, making it less relevant than functional validation.

D . Tests for each of the interfaces and process changes:

This is also essential. The project includes explicit "interface and process changes" alongside the migration. Interface updates (e.g., SAIL forms) might rely on new data structures or queries, while process changes (e.g., modified process models) could involve updated nodes or logic. Testing each change ensures these components function correctly with the new database and meet business requirements. Appian's testing guidelines emphasize targeted validation of modified components to confirm they integrate with the migrated data layer, making this a primary focus of the strategy.

E . Tests that ensure users can still successfully log into the platform:

Login testing verifies authentication (e.g., SSO, LDAP), typically managed by Appian's security layer, not the business database. A database migration affects application data, not user authentication, unless the database stores user credentials (uncommon in Appian, which uses separate identity management). While a quick sanity check, it's narrow and subsumed by broader regression testing (B), making it redundant as a standalone item.

Conclusion: The two key items are B (regression test of all existing system functionality) and D (tests for each of the interfaces and process changes). Regression testing (B) ensures the database migration doesn't disrupt the entire application, while targeted testing (D) validates the specific interface and process updates. Together, they cover the full scope-existing stability and new functionality-aligning with Appian's recommended approach for complex migrations and modifications.

Appian Documentation: "Testing Best Practices" (Regression and Component Testing).

Appian Lead Developer Certification: Application Maintenance Module (Database Migration Strategies).

Appian Best Practices: "Managing Large-Scale Changes in Appian" (Test Planning).

問題 #40

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 primary considerations when building this integration?

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

答案: B,C

解題說明:

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).

問題 #41

You are in a backlog refinement meeting with the development team and the product owner. You review a story for an integration involving a third-party system. A payload will be sent from the Appian system through the integration to the third-party system. The story is 21 points on a Fibonacci scale and requires development from your Appian team as well as technical resources from the third-party system. This item is crucial to your project's success. What are the two recommended steps to ensure this story can be developed effectively?

- A. Identify subject matter experts (SMEs) to perform user acceptance testing (UAT).
- B. Break down the item into smaller stories.
- C. Acquire testing steps from QA resources.
- D. Maintain a communication schedule with the third-party resources.

答案: B,D

解題說明:

Comprehensive and Detailed In-Depth Explanation:

This question involves a complex integration story rated at 21 points on the Fibonacci scale, indicating significant complexity and effort. Appian Lead Developer best practices emphasize effective collaboration, risk mitigation, and manageable development scopes for such scenarios. The two most critical steps are:

Option C (Maintain a communication schedule with the third-party resources):

Integrations with third-party systems require close coordination, as Appian developers depend on external teams for endpoint specifications, payload formats, authentication details, and testing support. Establishing a regular communication schedule ensures alignment on requirements, timelines, and issue resolution. Appian's Integration Best Practices documentation highlights the importance of proactive communication with external stakeholders to prevent delays and misunderstandings, especially for critical project components.

Option D (Break down the item into smaller stories):

A 21-point story is considered large by Agile standards (Fibonacci scale typically flags anything above 13 as complex). Appian's Agile Development Guide recommends decomposing large stories into smaller, independently deliverable pieces to reduce risk, improve testability, and enable iterative progress. For example, the integration could be split into tasks like designing the payload structure, building the integration object, and testing the connection-each manageable within a sprint. This approach aligns with the principle of delivering value incrementally while maintaining quality.

Option A (Acquire testing steps from QA resources): While QA involvement is valuable, this step is more relevant during the testing phase rather than backlog refinement or development preparation. It's not a primary step for ensuring effective development of the story.

Option B (Identify SMEs for UAT): User acceptance testing occurs after development, during the validation phase. Identifying SMEs is important but not a key step in ensuring the story is developed effectively during the refinement and coding stages.

By choosing C and D, you address both the external dependency (third-party coordination) and internal complexity (story size), ensuring a smoother development process for this critical integration.

