DUMPSARENA

Appian Certified Lead Developer

Appian ACD301

Version Demo

Total Demo Questions: 7

Total Premium Questions: 71

Buy Premium PDF

https://dumpsarena.co

sales@dumpsarena.co

sales@dumpsarena.co
dumpsarena.co

Topic Break Down

Topic	No. of Questions
Topic 1, Appian Design Guidance	11
Topic 2, Data Management	17
Topic 3, User Experience	5
Topic 4, Process Modeling	8
Topic 5, Integrations	16
Topic 6, Security	3
Topic 7, Deployment	9
Topic 8, Mix Questions	2
Total	71

QUESTION NO: 1

A development team is designing a reusable expression rule that will be used by several interfaces to retrieve active cases for a selected customer. The customer can have thousands of historical cases, but each interface only needs the case identifier, title, status, and opened date.

Which two practices should the team use when designing the rule? Select two.

A. Configure the rule to return only the fields required by the interfaces.

B. Accept the selected customer identifier as a rule input.

C. Return all historical cases so each interface can filter the result locally.

D. Reference the calling interface's local variables directly within the rule.

E. Use an unbounded batch size to avoid subsequent record queries.

ANSWER: A B

Explanation:

The rule should query only the fields needed by its consumers and should use paging appropriate to the interface use case. These practices reduce unnecessary data retrieval and make the rule suitable for reuse in multiple interfaces. The rule should also accept the customer identifier as an input rather than rely on an interface-specific local variable, allowing each consuming interface to provide its own context.

Returning every historical case without paging increases query cost and can create slow interfaces. Embedding a specific interface local variable in the rule makes the rule dependent on one calling context and prevents effective reuse.

QUESTION NO: 2

You need to design a complex Appian integration to call a RESTful API. The RESTful API will be used to update a case in a customer’s legacy system.

What are three prerequisites for designing the integration?

A. Define the HTTP method that the integration will use.

B. Understand the content of the expected body, including each field type and their limits.

C. Understand whether this integration will be used in an interface or in a process model.

D. Understand the different error codes managed by the API and the process of error handling in Appian.

E. Understand the business rules to be applied to ensure the business logic of the data.

ANSWER: A B D

Explanation:

“Define the HTTP method that the integration will use.” is required because the HTTP method expresses the operation that the endpoint performs. For an update operation, the API contract determines whether the request uses PUT, PATCH, POST, or another supported method; Appian’s HTTP integration configuration must match that contract.

“Understand the content of the expected body, including each field type and their limits.” is also fundamental. Before constructing an Appian request body, the developer needs the API’s required and optional fields, JSON structure, data types, permitted values, formatting requirements, and size or length constraints. This information enables reliable mapping from Appian data into the legacy system’s expected payload.

“Understand the different error codes managed by the API and the process of error handling in Appian.” is essential for a production-ready integration. The API’s documented response statuses and error payloads inform how the application handles validation failures, authentication issues, unavailable services, and successful updates. Appian integrations expose

response status information and can be designed with appropriate error handling, user feedback, and process-level recovery behavior. See Appian's HTTP Integration Object documentation and Integration Object documentation for configuration and response-handling concepts.

QUESTION NO: 3

A client plans to release an Appian application that will be used by users in several regions. A scheduled process creates a daily work item at 9:00 AM for each region, and the schedule must remain correct when daylight-saving changes occur.

Which two design considerations are appropriate? Select two.

A. Configure each regional schedule using the intended regional time zone.

B. Test scheduled executions around daylight-saving transitions.

C. Use the time zone of the user who completes the resulting task.

D. Convert every regional requirement to one fixed UTC offset for the entire year.

E. Republish the process model from a different user account for each region.

ANSWER: A B

Explanation:

The process scheduling design should use the intended regional time zone for each business schedule, and the team should test the schedules around daylight-saving transitions. A daily schedule evaluated in a correct regional time zone can account for local clock changes, while targeted testing validates that the configured process behavior matches the business expectation at the transition boundaries.

Using the time zone of whichever user completes the task does not control when an unattended scheduled process begins. Converting all requirements to one fixed offset can produce incorrect local execution times when daylight-saving time starts or ends. Publishing the model from different user accounts is not a reliable substitute for explicitly designing the intended schedule behavior.

QUESTION NO: 4

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

ANSWER: D

Explanation:

Provide Appian developers with the Designer permissions role within Appian, grant them only basic user rights outside their application, and assign them permissions to administer their application. This applies the principle of least privilege through Appian's platform and application security model. The Designer system role gives a developer the capability to build and maintain design objects, while application-specific administrator access scopes their administrative responsibility to the application they own. Outside that defined application boundary, the developer retains ordinary user-level access rather than broad design or administration privileges.

This arrangement prevents developers from inadvertently changing components in unrelated applications, which is the practical control needed to stop cross-application changes from creating unexpected behavior or broken dependencies. It also gives the responsible delivery team sufficient authority to maintain its own application without requiring broad platform-administration permissions. Appian security is evaluated at the object and application levels, so deliberately assigning system roles and application administrators is a durable technical safeguard, rather than relying solely on an informal team convention. Review Appian's guidance on user roles and object security for the distinction between platform capabilities and access to specific design objects and applications.

QUESTION NO: 5 - (SIMULATION)

For each scenario outlined, match the best tool to use to meet expectations. Each tool will be used once

Note: To change your responses, you may deselected your response by clicking the blank space at the top of the selection list.

ANSWER: See the explanation for the answer

Explanation:

Correct matching:

When a `Customer` update must make a field value appear on the `Company` Record List: **Database Complex View**.

When a `Customer` update requires a *simple* transformation on related `Customer` objects for the same company: **Database Trigger**.

When a `Customer` update requires *complex* transformations involving `Customer`, `Company`, and `Contract` objects: **Database Stored Procedure**.

When a `Customer` update requires *simple* transformations involving `Company`, `Address`, and `Contract` objects: **Write to Data Store Entity smart service**.

A complex database view is appropriate for presenting joined or derived data in a record list. A trigger handles lightweight, automatic database-side updates. A stored procedure is best for complex, multi-table transactional logic. The Write to Data Store Entity smart service is suitable when straightforward related-entity updates are managed in Appian.

QUESTION NO: 6

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?

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 processes.

D. Start the utility processes via a subprocess asynchronously.

ANSWER: C

Explanation:

Use Process Messaging to start the utility processes. Process messaging is appropriate when a process needs to initiate independent, fire-and-forget work and does not require a result to continue. The main process sends a message and can complete after its own retrieval and dispatch work, while each utility process is initiated by its configured message start

event. This creates a decoupled handoff: the utility work has its own process lifecycle and does not need to remain connected to the initiating process as a child subprocess.

This pattern is especially useful for an unattended, frequently executed integration process. Since no user task or utility-process output must be tracked by the main process, messaging avoids maintaining unnecessary parent-child process coordination. It also supports a clean event-driven design in which the receiving utility process defines the message it accepts and starts only when that message is received. The message payload can contain the identifiers or data needed by the utility process, keeping the initiating model focused on orchestration rather than downstream execution management.

Appian documents process messages as a mechanism for communication between process models, including starting a process through a message event. See the Appian Process Messaging documentation and the Appian Message Event documentation.

QUESTION NO: 7

You have 5 applications on your Appian platform in Production. Users are now beginning to use multiple applications across the platform, and the client wants to ensure a consistent user experience across all applications.

You notice that some applications use rich text, some use section layouts, and others use box layouts. The result is that each application has a different color and size for the header.

What would you recommend to ensure consistency across the platform?

A. Create constants for text size and color, and update each section to reference these values.

B. In the common application, create a rule that can be used across the platform for section headers, and update each application to reference this new rule.

C. In the common application, create one rule for each application, and update each application to reference its respective rule.

D. In each individual application, create a rule that can be used for section headers, and update each application to reference its respective rule.

ANSWER: B

Explanation:

In the common application, create a rule that can be used across the platform for section headers, and update each application to reference this new rule. This creates a shared, reusable interface component that centrally defines the section-header presentation, including the chosen layout component, text styling, color, and sizing. Each application can use the same rule expression wherever it needs a section header, so users see a consistent visual pattern as they move between applications.

A common application is an appropriate place for objects intended to be reused by multiple applications. Centralizing the header in one rule also improves maintainability: when the organization changes branding, accessibility styling, or header conventions, a developer updates the shared rule once and the change is inherited at every reference. This avoids duplicated UI definitions and helps ensure that later development follows the same established design pattern. Appian design guidance emphasizes reusable components as a way to standardize frequently used interface patterns and reduce repeated implementation effort. See Appian's Reusable Components documentation and Design Guidance documentation.