Examen ACD301: Desarrollo Avanzado en Appian

¿Qué es la certificación ACD301 de Appian?

La **certificación ACD301** de Appian es un examen que valida las habilidades y conocimientos de los desarrolladores en la plataforma de desarrollo Appian. Los candidatos deben demostrar competencia en áreas como el *desarrollo de aplicaciones*, *integración de datos* y *diseño de interfaces de usuario*. Esta certificación es esencial para quienes buscan avanzar en sus carreras como desarrolladores líderes en Appian.

¿Qué temas abarca el examen ACD301?

El examen ACD301 abarca una variedad de temas cruciales, incluyendo:

- **Fundamentos de la plataforma Appian**
- **Desarrollo de aplicaciones y flujos de trabajo**
- **Gestión y seguridad de registros**
- **Integración con sistemas externos**

Estos temas son esenciales para cualquier desarrollador que desee obtener la certificación y validar sus habilidades en Appian.

¿Cómo puedo prepararme para el examen ACD301?

Prepararse para el examen ACD301 implica una combinación de estudios teóricos y prácticos. Se recomienda participar en *capacitación oficial* de Appian, leer la *documentación de Appian* y practicar en proyectos reales. Además, es útil unirse a *comunidades en línea* para compartir experiencias y consejos con otros candidatos.

¿Cuál es la duración del examen ACD301?

El examen ACD301 típico dura **2 horas**. Durante este tiempo, los candidatos deben responder una serie de *preguntas de opción múltiple* que evalúan su comprensión y habilidades en la plataforma Appian. Es importante gestionar el tiempo de manera efectiva para asegurarse de que se responda a todas las preguntas.

¿Qué beneficios obtengo al estar certificado como Desarrollador Líder en Appian?

Obtener la certificación ACD301 proporciona varios beneficios, incluyendo mejoras en las oportunidades laborales, un aumento en la **credibilidad profesional** y la capacidad de liderar equipos de desarrollo en proyectos complejos. Además, la certificación puede aumentar su *salario* y hacerlo más atractivo para los empleadores que buscan **expertos en Appian**. Para más información, puedes visitar certkillers.net, una excelente fuente de recursos para tu preparación.

Appian

ACD301 Exam

Appian Certified Lead Developer

Thank you for Downloading ACD301 exam PDF Demo

You can buy Latest ACD301 Full Version Download

https://www.certkillers.net/Exam/ACD301

Version: 4.1

Question: 1

You are reviewing the Engine Performance Logs in Production for a single application that has been live for six months. This application experiences concurrent user activity and has a fairly sustained load during business hours. The client has reported performance issues with the application during business hours.
During your investigation, you notice a high Work Queue - Java Work Queue Size value in the logs. You also notice unattended process activities, including timer events and sending notification emails, are taking far longer to execute than normal.
The client increased the number of CPU cores prior to the application going live.
What is the next recommendation?

A. Add more engine replicas.
B. Optimize slow-performing user interfaces.
C. Add more application servers.
D. Add execution and analytics shards

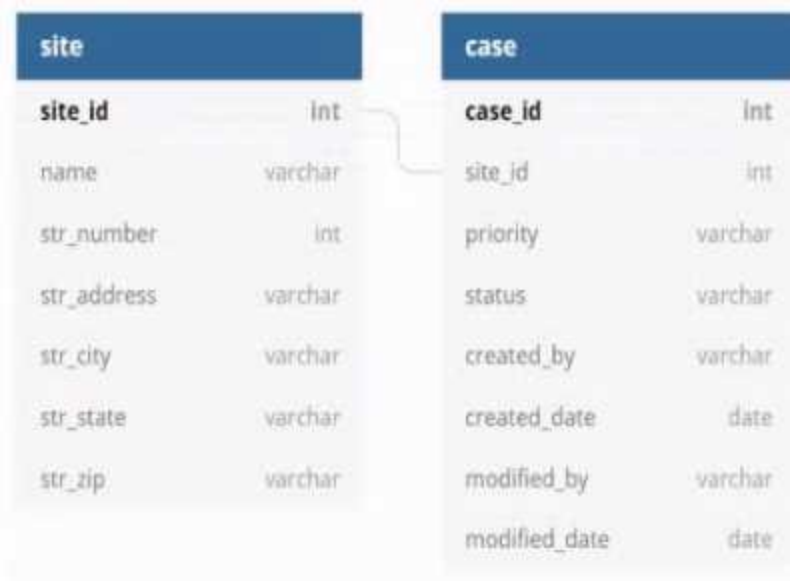
Answer: A

Explanation:
As an Appian Lead Developer, analyzing Engine Performance Logs to address performance issues in a Production application requires understanding Appian's architecture and the specific metrics described. The scenario indicates a high "Work Queue - Java Work Queue Size," which reflects a backlog of tasks in the Java Work Queue (managed by Appian engines), and delays in unattended process activities (e.g., timer events, email notifications). These symptoms suggest the Appian engines are overloaded, despite the client increasing CPU cores. Let's evaluate each option:
A . Add more engine replicas:
This is the correct recommendation. In Appian, engine replicas (part of the Appian Engine cluster) handle process execution, including unattended tasks like timers and notifications. A high Java Work Queue Size indicates the engines are overwhelmed by concurrent activity during business hours, causing delays. Adding more engine replicas distributes the workload, reducing queue size and improving performance for both user-driven and unattended tasks. Appian's documentation recommends scaling engine replicas to handle sustained loads, especially in Production with high concurrency. Since CPU cores were already increased (likely on application servers), the bottleneck is likely the engine capacity, not the servers.
B . Optimize slow-performing user interfaces:

While optimizing user interfaces (e.g., SAIL forms, reports) can improve user experience, the scenario highlights delays in unattended activities (timers, emails), not UI performance. The Java Work Queue Size issue points to engine-level processing, not UI rendering, so this doesn't address the root cause. Appian's performance tuning guidelines prioritize engine scaling for queue-related issues, making this a secondary concern.
C . Add more application servers:
Application servers handle web traffic (e.g., SAIL interfaces, API calls), not process execution or unattended tasks managed by engines. Increasing application servers would help with UI concurrency but wouldn't reduce the Java Work Queue Size or speed up timer/email processing, as these are engine responsibilities. Since the client already increased CPU cores (likely on application servers), this is redundant and unrelated to the issue.
D . Add execution and analytics shards:
Execution shards (for process data) and analytics shards (for reporting) are part of Appian's data fabric for scalability, but they don't directly address engine workload or Java Work Queue Size. Shards optimize data storage and query performance, not real-time process execution. The logs indicate an engine bottleneck, not a data storage issue, so this isn't relevant. Appian's documentation confirms shards are for long-term scaling, not immediate performance fixes.
Conclusion: Adding more engine replicas (A) is the next recommendation. It directly resolves the high Java Work Queue Size and delays in unattended tasks, aligning with Appian's architecture for handling concurrent loads in Production. This requires collaboration with system administrators to configure additional replicas in the Appian cluster.
Reference:
Appian Documentation: "Engine Performance Monitoring" (Java Work Queue and Scaling Replicas).
Appian Lead Developer Certification: Performance Optimization Module (Engine Scaling Strategies).
Appian Best Practices: "Managing Production Performance" (Work Queue Analysis).

Question: 2

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 column should be indexed?

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

Answer: ABE

Explanation:
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 Reference: 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.
Reference:
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).

Question: 3

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?

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.
Reference:
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: 4

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:
Comprehensive and Detailed In-Depth Explanation:
As an Appian Lead Developer, designing a complex integration to a RESTful API for updating a case in a legacy system requires a structured approach to ensure reliability, performance, and alignment with business needs. The integration involves sending a JSON payload (implied by the context) and handling responses, so the focus is on technical and functional prerequisites. Let's evaluate each option:
A . Define the HTTP method that the integration will use:
This is a primary prerequisite. RESTful APIs use HTTP methods (e.g., POST, PUT, GET) to define the operation—here, updating a case likely requires PUT or POST. Appian's Connected System and Integration objects require specifying the method to configure the HTTP request correctly. Understanding the API's method ensures the integration aligns with its design, making this essential for design. Appian's documentation emphasizes choosing the correct HTTP method as a foundational step.
B . Understand the content of the expected body, including each field type and their limits:
This is also critical. The JSON payload for updating a case includes fields (e.g., text, dates, numbers), and the API expects a specific structure with field types (e.g., string, integer) and limits (e.g., max length, size constraints). In Appian, the Integration object requires a dictionary or CDT to construct the body, and mismatches (e.g., wrong types, exceeding limits) cause errors (e.g., 400 Bad Request). Appian's best practices mandate understanding the API schema to ensure data compatibility, making this a key

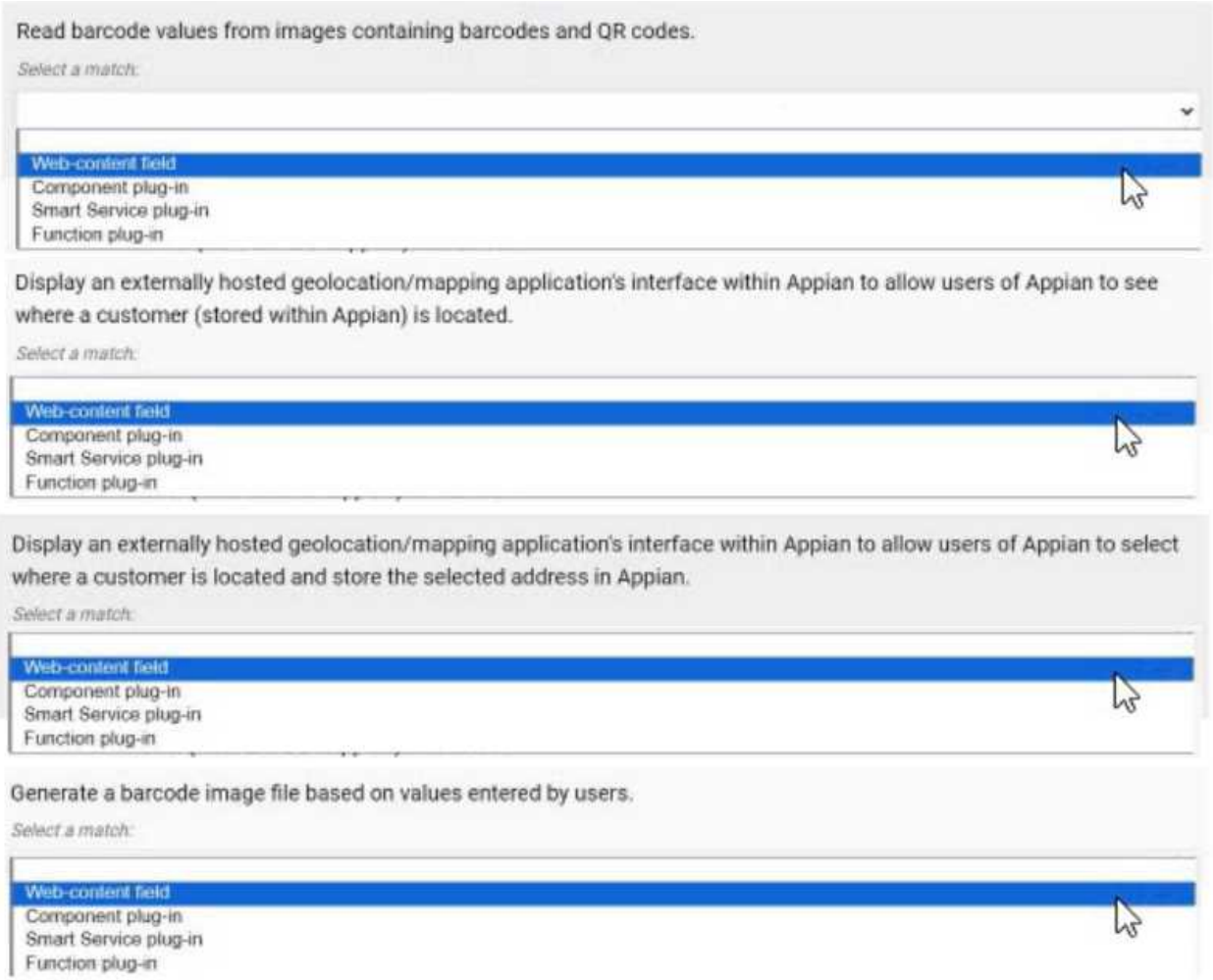
prerequisite.
C . Understand whether this integration will be used in an interface or in a process model:
While knowing the context (interface vs. process model) is useful for design (e.g., synchronous vs. asynchronous calls), it's not a prerequisite for the integration itself—it's a usage consideration. Appian supports integrations in both contexts, and the integration's design (e.g., HTTP method, body) remains the same. This is secondary to technical API details, so it's not among the top three prerequisites.
D . Understand the different error codes managed by the API and the process of error handling in Appian:
This is essential. RESTful APIs return HTTP status codes (e.g., 200 OK, 400 Bad Request, 500 Internal Server Error), and the customer's API likely documents these for failure scenarios (e.g., invalid data, server issues). Appian's Integration objects can handle errors via error mappings or process models, and understanding these codes ensures robust error handling (e.g., retry logic, user notifications). Appian's documentation stresses error handling as a core design element for reliable integrations, making this a primary prerequisite.
E . Understand the business rules to be applied to ensure the business logic of the data:
While business rules (e.g., validating case data before sending) are important for the overall application, they aren't a prerequisite for designing the integration itself—they're part of the application logic (e.g., process model or interface). The integration focuses on technical interaction with the API, not business validation, which can be handled separately in Appian. This is a secondary concern, not a core design requirement for the integration.
Conclusion: The three prerequisites are A (define the HTTP method), B (understand the body content and limits), and D (understand error codes and handling). These ensure the integration is technically sound, compatible with the API, and resilient to errors—critical for a complex RESTful API integration in Appian.
Reference:
Appian Documentation: "Designing REST Integrations" (HTTP Methods, Request Body, Error Handling).
Appian Lead Developer Certification: Integration Module (Prerequisites for Complex Integrations).
Appian Best Practices: "Building Reliable API Integrations" (Payload and Error Management).

To design a complex Appian integration to call a RESTful API, you need to have some prerequisites, such as:
Define the HTTP method that the integration will use. The HTTP method is the action that the integration will perform on the API, such as GET, POST, PUT, PATCH, or DELETE. The HTTP method determines how the data will be sent and received by the API, and what kind of response will be expected.
Understand the content of the expected body, including each field type and their limits. The body is the data that the integration will send to the API, or receive from the API, depending on the HTTP method. The body can be in different formats, such as JSON, XML, or form data. You need to understand how to structure the body according to the API specification, and what kind of data types and values are allowed for each field.
Understand the different error codes managed by the API and the process of error handling in Appian. The error codes are the status codes that indicate whether the API request was successful or not, and what kind of problem occurred if not. The error codes can range from 200 (OK) to 500 (Internal Server Error), and each code has a different meaning and implication. You need to understand how to handle different error codes in Appian, and how to display meaningful messages to the user or log them for debugging purposes.
The other two options are not prerequisites for designing the integration, but rather considerations for implementing it.
Understand whether this integration will be used in an interface or in a process model. This is not a prerequisite, but rather a decision that you need to make based on your application requirements and

design. You can use an integration either in an interface or in a process model, depending on where you need to call the API and how you want to handle the response. For example, if you need to update a case in real-time based on user input, you may want to use an integration in an interface. If you need to update a case periodically based on a schedule or an event, you may want to use an integration in a process model.
Understand the business rules to be applied to ensure the business logic of the data. This is not a prerequisite, but rather a part of your application logic that you need to implement after designing the integration. You need to apply business rules to validate, transform, or enrich the data that you send or receive from the API, according to your business requirements and logic. For example, you may need to check if the case status is valid before updating it in the legacy system, or you may need to add some additional information to the case data before displaying it in Appian.

Question: 5

HOTSPOT
For each requirement, match the most appropriate approach to creating or utilizing plug-ins Each approach will be used once.
Note: To change your responses, you may deselect your response by clicking the blank space at the top of the selection list.



Answer:

Explanation:

Read barcode values from images containing barcodes and QR codes. → Smart Service plug-in

Display an externally hosted geolocation/mapping application's interface within Appian to allow users of Appian to see where a customer (stored within Appian) is located. → Web-content field

Display an externally hosted geolocation/mapping application's interface within Appian to allow users of Appian to select where a customer is located and store the selected address in Appian. → Component plug-in

Generate a barcode image file based on values entered by users. → Function plug-in

Comprehensive and Detailed In-Depth Explanation:
Appian plug-ins extend functionality by integrating custom Java code into the platform. The four approaches—Web-content field, Component plug-in, Smart Service plug-in, and Function plug-in—serve distinct purposes, and each requirement must be matched to the most appropriate one based on its use case. Appian's Plug-in Development Guide provides the framework for these decisions.

Read barcode values from images containing barcodes and QR codes → Smart Service plug-in:
This requirement involves processing image data to extract barcode or QR code values, a task that typically occurs within a process model (e.g., as part of a workflow). A Smart Service plug-in is ideal because it allows custom Java logic to be executed as a node in a process, enabling the decoding of images and returning the extracted values to Appian. This approach integrates seamlessly with Appian's process automation, making it the best fit for data extraction tasks.

Display an externally hosted geolocation/mapping application's interface within Appian to allow users of Appian to see where a customer (stored within Appian) is located → Web-content field:
This requires embedding an external mapping interface (e.g., Google Maps) within an Appian interface. A Web-content field is the appropriate choice, as it allows you to embed HTML, JavaScript, or iframe content from an external source directly into an Appian form or report. This approach is lightweight and does not require custom Java development, aligning with Appian's recommendation for displaying external content without interactive data storage.

Display an externally hosted geolocation/mapping application's interface within Appian to allow users of Appian to select where a customer is located and store the selected address in Appian → Component plug-in:
This extends the previous requirement by adding interactivity (selecting an address) and data storage. A Component plug-in is suitable because it enables the creation of a custom interface component (e.g., a map selector) that can be embedded in Appian interfaces. The plug-in can handle user interactions, communicate with the external mapping service, and update Appian data stores, offering a robust solution for interactive external integrations.

Generate a barcode image file based on values entered by users → Function plug-in:
This involves generating an image file dynamically based on user input, a task that can be executed within an expression or interface. A Function plug-in is the best match, as it allows custom Java logic to

be called as an expression function (e.g., pluginGenerateBarcode(value)), returning the generated image. This approach is efficient for single-purpose operations and integrates well with Appian's expression-based design.

Matching Rationale:

Each approach is used once, as specified, covering the spectrum of plug-in types: Smart Service for process-level tasks, Web-content field for static external display, Component plug-in for interactive components, and Function plug-in for expression-level operations.

Appian's plug-in framework discourages overlap (e.g., using a Smart Service for display or a Component for process tasks), ensuring the selected matches align with intended use cases.

Reference: Appian Documentation - Plug-in Development Guide, Appian Interface Design Best Practices, Appian Lead Developer Training - Custom Integrations.

Thank You for trying ACD301 PDF Demo

To Buy New ACD301 Full Version Download visit link below

https://www.certkillers.net/Exam/ACD301

Start Your ACD301 Preparation

[Limited Time Offer] Use Coupon "CKNET" for Further discount on your purchase. Test your ACD301 preparation with actual exam questions.