Free Questions for MCPA-Level-1-Maintenance by certsinside

Shared by Finley on 29-01-2024

For More Free Questions and Preparation Resources

Check the Links on Last Page

Question 1

Question Type: MultipleChoice

A company uses a hybrid Anypoint Platform deployment model that combines the EU control plane with customer-hosted Mule runtimes. After successfully testing a Mule API implementation in the Staging environment, the Mule API implementation is set with environment-specific properties and must be promoted to the Production environment. What is a way that MuleSoft recommends to configure the Mule API implementation and automate its promotion to the Production environment?

Options:

A- Bundle properties files for each environment into the Mule API implementation's deployable archive, then promote the Mule API implementation to the Production environment using Anypoint CLI or the Anypoint Platform REST APIsB.

B- Modify the Mule API implementation's properties in the API Manager Properties tab, then promote the Mule API implementation to the Production environment using API Manager

C- Modify the Mule API implementation's properties in Anypoint Exchange, then promote the Mule API implementation to the Production environment using Runtime Manager

D- Use an API policy to change properties in the Mule API implementation deployed to the Staging environment and another API policy to deploy the Mule API implementation to the Production environment

Answer:

A

Explanation:

Correct Answer: Bundle properties files for each environment into the Mule API

****

>> Anypoint Exchange is for asset discovery and documentation. It has got no provision to modify the properties of Mule API implementations at all.

>> API Manager is for managing API instances, their contracts, policies and SLAs. It has also got no provision to modify the properties of API implementations.

>> API policies are to address Non-functional requirements of APIs and has again got no provision to modify the properties of API implementations.

So, the right way and recommended way to do this as part of development practice is to bundle properties files for each environment into the Mule API implementation and just point and refer to respective file per environment.

Question 2

Question Type: MultipleChoice

A system API is deployed to a primary environment as well as to a disaster recovery (DR) environment, with different DNS names in each environment. A process API is a client to the system API and is being rate limited by the system API, with different limits in each of the environments. The system API's DR environment provides only 20% of the rate limiting offered by the primary environment. What is the best API fault-tolerant invocation strategy to reduce overall errors in the process API, given these conditions and constraints?

Options:

A- Invoke the system API deployed to the primary environment; add timeout and retry logic to the process API to avoid intermittent failures; if it still fails, invoke the system API deployed to the DR environment

B- Invoke the system API deployed to the primary environment; add retry logic to the process API to handle intermittent failures by invoking the system API deployed to the DR environment

C- In parallel, invoke the system API deployed to the primary environment and the system API deployed to the DR environment; add timeout and retry logic to the process API to avoid intermittent failures; add logic to the process API to combine the results

D- Invoke the system API deployed to the primary environment; add timeout and retry logic to the process API to avoid intermittent failures; if it still fails, invoke a copy of the process API deployed to the DR environment

Answer:

A

Explanation:

Correct Answer: Invoke the system API deployed to the primary environment; add timeout

****

There is one important consideration to be noted in the question which is - System API in DR environment provides only 20% of the rate limiting offered by the primary environment. So, comparitively, very less calls will be allowed into the DR environment API opposed to its primary environment. With this in mind, lets analyse what is the right and best fault-tolerant invocation strategy.

1. Invoking both the system APIs in parallel is definitely NOT a feasible approach because of the 20% limitation we have on DR environment. Calling in parallel every time would easily and quickly exhaust the rate limits on DR environment and may not give chance to genuine intermittent error scenarios to let in during the time of need.

2. Another option given is suggesting to add timeout and retry logic to process API while invoking primary environment's system API. This is good so far. However, when all retries failed, the option is suggesting to invoke the copy of process API on DR environment which is not right or recommended. Only system API is the one to be considered for fallback and not the whole process API. Process APIs usually have lot of heavy orchestration calling many other APIs which we do not want to repeat again by calling DR's process API. So this option is NOT right.

3. One more option given is suggesting to add the retry (no timeout) logic to process API to directly retry on DR environment's system API instead of retrying the primary environment system API first. This is not at all a proper fallback. A proper fallback should occur only after all retries are performed and exhausted on Primary environment first. But here, the option is suggesting to directly retry fallback API on first failure itself without trying main API. So, this option is NOT right too.

This leaves us one option which is right and best fit.

- Invoke the system API deployed to the primary environment

- Add Timeout and Retry logic on it in process API

- If it fails even after all retries, then invoke the system API deployed to the DR environment.

Question 3

Question Type: MultipleChoice

In which layer of API-led connectivity, does the business logic orchestration reside?

Options:

A- System Layer

B- Experience Layer

C- Process Layer

Answer:

C

Explanation:

Correct Answer: Process Layer

****

>> Experience layer is dedicated for enrichment of end user experience. This layer is to meet the needs of different API clients/ consumers.

>> System layer is dedicated to APIs which are modular in nature and implement/ expose various individual functionalities of backend systems

>> Process layer is the place where simple or complex business orchestration logic is written by invoking one or many System layer modular APIs

So, Process Layer is the right answer.

Question 4

Question Type: MultipleChoice

Once an API Implementation is ready and the API is registered on API Manager, who should request the access to the API on Anypoint Exchange?

Options:

A- None

B- Both

C- API Client

D- API Consumer

Answer:

D

Explanation:

Correct Answer: API Consumer

****

>> API clients are piece of code or programs that use the client credentials of API consumer but does not directly interact with Anypoint Exchange to get the access

>> API consumer is the one who should get registered and request access to API and then API client needs to use those client credentials to hit the APIs

So, API consumer is the one who needs to request access on the API from Anypoint Exchange

Question 5

Question Type: MultipleChoice

Traffic is routed through an API proxy to an API implementation. The API proxy is managed by API Manager and the API implementation is deployed to a CloudHub VPC using Runtime Manager. API policies have been applied to this API. In this deployment scenario, at what point are the API policies enforced on incoming API client requests?

Options:

A- At the API proxy

B- At the API implementation

C- At both the API proxy and the API implementation

D- At a MuleSoft-hosted load balancer

Answer:

A

Explanation:

Correct Answer: At the API proxy

****

>> API Policies can be enforced at two places in Mule platform.

>> One - As an Embedded Policy enforcement in the same Mule Runtime where API implementation is running.

>> Two - On an API Proxy sitting in front of the Mule Runtime where API implementation is running.

>> As the deployment scenario in the question has API Proxy involved, the policies will be enforced at the API Proxy.

Question 6

Question Type: MultipleChoice

An API client calls one method from an existing API implementation. The API implementation is later updated. What change to the API implementation would require the API client's invocation logic to also be updated?

Options:

A- When the data type of the response is changed for the method called by the API client

B- When a new method is added to the resource used by the API client

C- When a new required field is added to the method called by the API client

D- When a child method is added to the method called by the API client

Answer:

C

Explanation:

Correct Answer: When a new required field is added to the method called by the API client

****

>> Generally, the logic on API clients need to be updated when the API contract breaks.

>> When a new method or a child method is added to an API , the API client does not break as it can still continue to use its existing method. So these two options are out.

>> We are left for two more where 'datatype of the response if changed' and 'a new required field is added'.

>> Changing the datatype of the response does break the API contract. However, the question is insisting on the 'invocation' logic and not about the response handling logic. The API client can still invoke the API successfully and receive the response but the response will have a different datatype for some field.

>> Adding a new required field will break the API's invocation contract. When adding a new required field, the API contract breaks the RAML or API spec agreement that the API client/API consumer and API provider has between them. So this requires the API client invocation logic to also be updated.

Question 7

Question Type: MultipleChoice

An organization has created an API-led architecture that uses various API layers to integrate mobile clients with a backend system. The backend system consists of a number of specialized components and can be accessed via a REST API. The process and experience APIs share the same bounded-context model that is different from the backend data model. What additional canonical models, bounded-context models, or anti-corruption layers are best added to this architecture to help process data consumed from the backend system?

Options:

A- Create a bounded-context model for every layer and overlap them when the boundary contexts overlap, letting API developers know about the differences between upstream and downstream data models

B- Create a canonical model that combines the backend and API-led models to simplify and unify data models, and minimize data transformations.

C- Create a bounded-context model for the system layer to closely match the backend data model, and add an anti-corruption layer to let

the different bounded contexts cooperate across the system and process layers

D- Create an anti-corruption layer for every API to perform transformation for every data model to match each other, and let data simply travel between APIs to avoid the complexity and overhead of building canonical models

Answer:

C

Explanation:

Correct Answer: Create a bounded-context model for the system layer to closely match the

****

>> Canonical models are not an option here as the organization has already put in efforts and created bounded-context models for Experience and Process APIs.

>> Anti-corruption layers for ALL APIs is unnecessary and invalid because it is mentioned that experience and process APIs share same bounded-context model. It is just the System layer APIs that need to choose their approach now.

>> So, having an anti-corruption layer just between the process and system layers will work well. Also to speed up the approach, system APIs can mimic the backend system data model.