PassQueen
a.k.a dumpsqueen.com

# Google Cloud - Apigee Certified API Engineer

## Google Apigee-API-Engineer

**Version Demo**

**Total Demo Questions: 16**

**Total Premium Questions: 162**

**Buy Premium PDF**

https://passqueen.com

support@passqueen.com

# Topic Break Down

| Topic | No. of Questions |
|---|---|
| Topic 1, Designing APIs | 44 |
| Topic 2, Developing APIs | 95 |
| Topic 3, Deploying and managing APIs | 15 |
| Topic 4, Monitoring and troubleshooting APIs | 7 |
| Topic 5, Mix Questions | 1 |
| Total | 162 |

QUESTION NO: 1

Which feature can be used to automatically distribute traffic across multiple target servers'?

**A.** use a concurrent rate limiting policy

**B.** use a LoadBalancer entry in the HTTPTargetConnection element

**C.** use RouteRules with multiple TargetEndpoints

**D.** use an AssignMessage policy

**ANSWER: B**

**Explanation:**

The **LoadBalancer** entry in the **HTTPTargetConnection** configuration is the Apigee feature designed to distribute API proxy traffic across multiple backend target servers. A TargetEndpoint can define several server endpoints within its LoadBalancer configuration, allowing Apigee to select a backend for each request instead of directing all traffic to one fixed URL. This supports backend availability and traffic distribution while keeping the target-server details managed in the API proxy's target configuration. Apigee supports load-balancing algorithms such as round robin, weighted, and least connections, so the distribution behavior can be selected to fit the backend topology and capacity. Target servers can be represented by named TargetServer resources, which centralize host, port, and TLS settings and can then be referenced by the load balancer. This is the appropriate configuration-level capability for automatically routing requests among multiple backend servers. See the Apigee load balancing across backend servers documentation and the TargetEndpoint and HTTPTargetConnection reference for the configuration structure and supported load-balancing behavior.

**QUESTION NO: 2**

A backend service returns XML, but mobile clients require JSON. Which Apigee policy should you use to convert the backend response payload to JSON?

**A.** XMLToJSON

**B.** JSONToXML

**C.** VerifyAPIKey

**D.** RegularExpressionProtection

**ANSWER: A**

**Explanation:**

**XMLToJSON** is the policy designed to convert an XML message payload into JSON. It can be attached in the target response or proxy response flow after the backend response is available. The policy transforms the XML representation so that the client receives a JSON response.

The proxy should also set the appropriate response `Content-Type`, normally `application/json`, when returning the converted payload. If only a subset of the XML document is required, an ExtractVariables policy can first extract the relevant XML fragment before conversion. See the XMLToJSON policy reference.

**QUESTION NO: 3**

Which of the following properties have type as "required" in the OpenAPI Specification 2.0?

Choose 3 answers

**A.** Info

**B.** host

**C.** paths

**D.** license

**E.** swagger

**F.** basePath

**G.** produces

**H.** consumes

**ANSWER: A C E**

**Explanation:**

In an OpenAPI Specification 2.0 document, the top-level Swagger Object has three mandatory fixed fields: `swagger`, `info`, and `paths`. The `swagger` property identifies the specification version used by the document and must contain the fixed value `"2.0"`. This lets tools recognize and parse the API definition according to the OpenAPI 2.0 format. The `info` property is required because it provides the API metadata through an Info Object, including required API title and version details. The `paths` property is required because it contains the Paths Object that defines the API's available endpoints and their operations. A valid document may contain an empty Paths Object when no operations are described yet, but the property itself must still be present. These requirements are defined in the Swagger Object section of the OpenAPI 2.0 specification. See the OpenAPI Specification version 2.0 and the corresponding OpenAPI Initiative 2.0 specification source.

## QUESTION NO: 4

You have a single back end that needs to be exposed to customers using different API request and response payloads You need to allow these different request types without breaking existing implementations What should you do?

**A.** Create a new API proxy for new customers and invoke backend target system with required parameters.

**B.** Configure the API as a pass-through proxy and invoke backend target system with client request parameters.

**C.** Create a new proxy xml and base path with upgraded version and invoke backend target system with required parameters

**D.** Include a new customer requirement in an existing API proxy and invoke backend target system with required parameters.

**ANSWER: C**

**Explanation:**

Create a new proxy xml and base path with upgraded version and invoke backend target system with required parameters is correct because API versioning lets the organization introduce a contract with new request and response payloads while preserving the established contract for existing consumers. The new version can have its own base path, such as `/v2`, and independently apply Apigee policies to validate, transform, and map its payload into the format required by the shared backend. Existing clients continue to call the earlier base path and receive the request and response structure they already support, avoiding a breaking change.

Apigee API proxies provide a boundary between client-facing API contracts and backend implementations. This boundary is specifically useful when multiple client experiences must be supported by one backend: each versioned proxy can expose its own resources, payload model, policies, and flows, then invoke the same target service with the appropriate backend parameters. Google recommends including the API version in the base path when versioning is required, which makes the version explicit in the consumer-facing URL and enables both versions to coexist during a controlled migration. See Manage APIs and Understanding APIs and API proxies.

## QUESTION NO: 5

Your API generates tokens to authenticate users. You have the following requirements

1. Limited token lifetime.

2. Managed key rotation.

3. Self-verifiable content.

4 Compact data representation

5. Refresh without new challenge.

You plan to use SAML2 Which two of the above-listed requirements are satisfied by using

SAML2? Choose 2 answers

**A.** Limited token lifetime.

**B.** Managed key rotation

**C.** Self-verifiable content

**D.** Compact data representation

**E.** Refresh without a new challenge

**ANSWER: A**

**Explanation:**

Limited token lifetime is supported by SAML 2.0 because a SAML assertion can contain validity constraints in its `Conditions` element. In particular, an identity provider can set the `NotBefore` and `NotOnOrAfter` attributes to define the time window during which a service provider may accept the assertion. A service provider validating the assertion checks these conditions, along with the assertion audience and other relevant constraints, before granting access. This enables an API architecture to ensure that an issued authentication assertion is usable only for a deliberately short period, reducing exposure if the assertion is intercepted or otherwise compromised. The exact validity duration is a deployment policy decision, but the SAML specification supplies the standard assertion fields needed to enforce expiry. Google Cloud Identity's SAML implementation likewise requires assertions to specify an expiry time through `NotOnOrAfter`. See the OASIS SAML 2.0 Core specification and Google Workspace SAML SSO documentation.

## QUESTION NO: 6

As an API Engineer your team would like to make sure you are simulating a user experience prior to a deployment in a production environment. Which tests should be ran to closely resemble a consumer interaction with a APIs?

**A.** Unit tests

**B.** Smoke tests.

**C.** Integration tests

**D.** Code quality analysis

**ANSWER: C**

**Explanation:**

Integration tests are the appropriate choice because they validate an API as a consumer would use it: by sending requests through the deployed proxy and exercising the connections among API proxy logic, target services, authentication, policies, data transformations, and returned responses. This provides confidence that the components work together under realistic request and response conditions before a production deployment. In Apigee, automated API testing can execute requests against an environment and assert expected status codes, headers, payload content, and other behavioral outcomes. Those assertions make it possible to verify the complete consumer-facing contract rather than only isolated implementation details.

A well-designed integration test suite should use representative request paths, methods, credentials, parameters, and payloads that match intended client behavior, while validating the resulting API behavior end to end. This makes integration tests particularly valuable in a pre-production environment, where the team can identify issues arising from interactions between the proxy and its dependencies before actual consumers encounter them. See Google Cloud's guidance on continuous integration for Apigee and the Apigee documentation for testing API proxies.

## QUESTION NO: 7

Your team has the following requirements in building an API:

• Adhere to Pragmatic REST.

• Model the API to the consumption use case.

• Require API Key authentication

• Implement CORS

• Validate inputs.

You have begun migrating a SOAP-based web service to a REST API by using the SOAP to REST function in Apigee Edge Which two of the above-listed requirements could be satisfied by this action?

Choose 2 answers

**A.** Adhere to Pragmatic REST

**B.** Model the API to the consumption use case

**C.** Require API Key authentication

**D.** Implement CORS.

**E.** Validate inputs.

## ANSWER: B C

**Explanation:**

Using Apigee Edge's SOAP-to-REST proxy creation capability can satisfy the requirements to **model the API to the consumption use case** and to **require API Key authentication**. The SOAP proxy wizard imports a WSDL and exposes selected SOAP operations through REST-facing resources. During proxy creation, the API designer can define the REST base path, resource paths, HTTP verbs, and request parameters that application developers consume, rather than exposing the SOAP endpoint directly. This gives the API a consumer-oriented REST interface while Apigee transforms the request and invokes the underlying SOAP service.

The generated proxy can also be configured with API-key verification. Apigee API keys identify the calling application and are validated by the proxy using the Verify API Key policy; valid keys are associated with developer apps and API products. This enables an API key requirement at the API proxy boundary while retaining the SOAP implementation behind it. See the Apigee documentation on creating a SOAP API proxy and the Verify API Key policy.

## QUESTION NO: 8

Which of the following properties have type as " required " in the OpenAPI Specification 2.0? Choose 3 answers

**A.** Info

**B.** host

**C.** paths

**D.** license

**E.** swagger

**F.** basePath

**G.** produces

**ANSWER: A C E**

**Explanation:**

In an OpenAPI Specification 2.0 document, the top-level required fields are `swagger`, `info`, and `paths`. The `swagger` field identifies the version of the Swagger/OpenAPI specification used by the document; for OpenAPI 2.0, its value must be `"2.0"`. This declaration lets tooling interpret the document according to the appropriate specification version.

The `info` field supplies required API metadata through an Info Object. Although the Info Object itself has its own required members, including `title` and `version`, the top-level API document must include the `info` property to provide that metadata. The `paths` field is also required because it contains the available API paths and their operations. It can be an empty object when no operations are exposed, but it must still be present in a valid OpenAPI 2.0 definition. These requirements are defined in the OpenAPI 2.0 root Swagger Object documentation: OpenAPI Specification 2.0 and the canonical specification source at OpenAPI Initiative GitHub repository.

## QUESTION NO: 9

As an API Engineer you get a calendar invite for a backlog grooming session. What should you expect to take place in the meeting?

**A.** Review the Epics and meet the cross functional team

**B.** Review and update the user stories and tasks in detail.

**C.** Review the high level design document and ask questions

**D.** Update support tickets that have been sitting for x number of days

**ANSWER: B**

**Explanation:**

“Review and update the user stories and tasks in detail.” is the expected activity in a backlog grooming session, more commonly called backlog refinement. This is a collaborative working session in which the delivery team and product stakeholders make upcoming backlog items ready for future sprint planning. For API work, that commonly means clarifying a user story’s intended API behavior, acceptance criteria, dependencies, security requirements, error handling, and any implementation or testing work needed to deliver it. The group may split an item that is too large, identify missing details, update its priority, and estimate the effort once the work is sufficiently understood. The practical outcome is a healthier, better-understood, and appropriately ordered backlog so that the team can select actionable work during Sprint Planning. Although the Scrum Guide does not prescribe a separate meeting called “grooming,” it defines Product Backlog refinement as the ongoing activity of breaking down and further defining backlog items, including adding description, order, and size. This matches a detailed review and update of user stories and associated work. See the Scrum Guide and Atlassian’s overview of backlog refinement.

## QUESTION NO: 10

Which of the following statements are true for the out-of-the-box Apigee Edge Analytics

Dashboard? Select all that apply Choose 3 answers

**A.** Visualize API proxy error metrics

**B.** Visualize API traffic metrics

**C.** Visualize Developer Engagement metrics

**D.** Visualize API Deployment metrics

**E.** Visualize Management API traffic metrics

**ANSWER: A B C**

**Explanation:**

Out-of-the-box Apigee Edge Analytics dashboards provide operational and product-adoption visibility for API proxies. **Visualize API proxy error metrics** is correct because the Errors dashboard reports API-proxy failures, including error rates, error counts, fault codes, and the proxies or target services associated with errors. This enables API teams to identify reliability issues and investigate failing traffic.

**Visualize API traffic metrics** is correct because the Traffic dashboard presents request volume and related measures over time. Teams can use these metrics to understand API usage patterns, assess proxy activity, and filter analytics data by dimensions such as proxy, developer app, product, and environment.

**Visualize Developer Engagement metrics** is correct because the Engagement dashboard is intended to show how developers and developer applications consume published APIs. It provides visibility into API-product and app usage, helping organizations evaluate adoption and engagement across their API program.

These standard dashboards are part of Apigee's built-in analytics experience and are designed around API runtime traffic and consumption data. See the Apigee Analytics dashboards documentation and the Apigee analytics overview for dashboard capabilities and analytics concepts.

**QUESTION NO: 11**

Which is true about DefaultFaultRule"?

**A.** DefaultFaultRule can be used to handle an error that is not explicitly handled by a FaultRule.

**B.** DefaultFaultRules are a fixed set of fault rules defined in the product

**C.** You cannot use both FaultRule and DefaultFaultRules in the same API proxy

**D.** DefaultFaultRule is used when the fault happens during communication with the target

**ANSWER: A**

**Explanation:**

`DefaultFaultRule` can provide a proxy-wide fallback for faults that are not handled by a more specific `FaultRule`. In Apigee, fault rules are evaluated when an error occurs during proxy processing. If no conditional `FaultRule` is selected for that fault, Apigee runs the `DefaultFaultRule`, allowing the proxy to return a consistent error payload, set headers, log details, or apply other common error-handling steps. This makes it useful for defining a reliable default response for unexpected faults without duplicating the same policies in every individual fault condition.

A `DefaultFaultRule` is defined in the API proxy endpoint configuration and can contain a sequence of policies, just as a regular fault rule can. Apigee also supports the `AlwaysEnforce` element for a default fault rule when behavior requires it to run even when another fault rule has been executed. See the Apigee fault handling documentation and the DefaultFaultRule configuration reference.

**QUESTION NO: 12**

Your APIs are configured as a relying party on an OpenID Connect platform. You need to inspect and verify the OpenID Connect identity. What two actions should you take?

Choose 2 answers

**A.** Verify the signature of the JWT using a shared secret.

**B.** Parse the JWT to extract the exp: nbf and iat properties to determine if the token is still valid

**C.** Pass the JWT to a preconfigured 3rd party for verification of the signature, exp, nbf and iat properties

**D.** Use the OpenID Connect URL to locate a trusted 3rd party for verification the signature, exp, nbf and iat properties

**E.** Using the JWKS URL in the OpenID Connect configuration, fetch the signing key to verify the JWT signature and parameters

**ANSWER: B E**

**Explanation:**

An OpenID Connect ID token is a JWT whose claims and signature must both be validated before the relying party can trust the authenticated identity. “Parse the JWT to extract the exp: nbf and iat properties to determine if the token is still valid” is required because these registered claims establish the token’s permitted validity period: `exp` limits when it may be accepted, `nbf` identifies when it becomes valid, and `iat` records when it was issued. Validation should account for an appropriately small clock-skew allowance where needed.

“Using the JWKS URL in the OpenID Connect configuration, fetch the signing key to verify the JWT signature and parameters” supplies the cryptographic verification required for an ID token received from an OpenID Connect provider. The provider’s discovery metadata publishes a `jwks_uri`; the relying party obtains the provider’s public signing keys from that endpoint, selects the key matching the JWT header’s `kid`, and verifies the signature. In Apigee, the VerifyJWT policy can verify a signed JWT and validate relevant claims as part of API request processing. See the OpenID Connect Discovery specification and Google Cloud’s VerifyJWT policy documentation.

## QUESTION NO: 13

When is it appropriate to use query parameters in RESTful API design? Select all that are correct

**A.** When passing username and passwords.

**B.** When providing the ability to return different levels of detail in the response.

**C.** When requesting that an entire collection be deleted.

**D.** When filtering the response based upon a query

**ANSWER: B D**

**Explanation:**

Query parameters are appropriate for expressing optional controls over how a resource representation is returned, without changing the identity of the resource path being addressed. “When providing the ability to return different levels of detail in the response.” is correct because APIs commonly use parameters such as `fields`, `view`, `expand`, or `include` to let clients select a representation or related data appropriate to their use case. This can reduce unnecessary payload size while retaining a stable resource-oriented URI structure.

“When filtering the response based upon a query” is also correct. Collection endpoints commonly accept query parameters to select the subset of resources a client needs, such as `/orders?status=shipped` or `/products?category=books`. Query parameters are also a natural mechanism for related collection operations such as pagination, sorting, and search. Google’s API design guidance describes using query parameters for standard list-method controls, including filtering and selecting a response view, while its resource-oriented design guidance distinguishes these optional request controls from the resource names in the URL path. These practices make APIs more efficient and predictable for clients while preserving clear resource paths. See Google Cloud API Design Guide: List pagination and Google Cloud API Design Guide: List filtering.

## QUESTION NO: 14

You have configured a TargetEndpoint with several backend servers. One server becomes unavailable, and Apigee must stop routing requests to that server until it is healthy again. What should you configure?

**A.** Configure a HealthMonitor in the LoadBalancer

**B.** Add a SpikeArrest policy to the TargetEndpoint

**C.** Add a RaiseFault policy to the ProxyEndpoint

**D.** Configure a RouteRule for every backend server

**ANSWER: A**

**Explanation:**

Configure a **HealthMonitor** in the TargetEndpoint LoadBalancer configuration. A health monitor sends periodic requests to each configured target server and evaluates the response against the configured success criteria. When a server fails the health checks, Apigee marks it unhealthy and stops selecting it for new requests.

When the server passes health checks again, it can be returned to the available server pool. Load balancing alone distributes requests, but a health monitor provides the active health information required to avoid an unavailable target. See Load balancing across backend servers.

## QUESTION NO: 15

Which practices help protect an API proxy from malicious JSON request payloads? Select all that are correct

**A.** Use JSONThreatProtection to limit JSON nesting depth.

**B.** Use JSONThreatProtection to limit array-element counts.

**C.** Validate required fields and permitted values before calling the backend.

**D.** Use VerifyAPIKey to validate the JSON schema.

**E.** Increase the response cache TimeToLive.

**ANSWER: A B C**

**Explanation:**

The JSONThreatProtection policy can enforce limits on JSON payload complexity, such as maximum nesting depth, object-entry count, array-element count, and string length. Applying appropriate limits helps prevent excessively complex JSON documents from consuming proxy or backend processing resources.

Input validation should also verify that required fields have expected data types, formats, and permitted values before the request reaches the backend. An API key identifies an application but does not validate the structure or complexity of a JSON payload. Increasing a cache TTL has no role in protecting against malicious request bodies. See the JSONThreatProtection policy reference and Apigee API security documentation.

## QUESTION NO: 16

Which are NOT a step in the OAuth 2 0 authorization code grant process? Select all that are correct

**A.** generate an authorization code

**B.** verify the device ID

**C.** validate the client API key

**D.** obtain the end user's consent for the application to request the user's protected resources

**E.** validate the developer name

**ANSWER: B E**

**Explanation:**

Verifying a device ID is not a defined step in the OAuth 2.0 authorization code grant. This grant is designed around a resource owner authorizing a client, an authorization server issuing a short-lived authorization code, and the client redeeming that code at the token endpoint. The protocol associates the request with the client and its registered redirect URI; it does not require a device identifier as part of the standard authorization-code exchange.

Validating the developer name is also not an OAuth 2.0 authorization code grant step. In Apigee, a developer is an entity used to organize and own apps, credentials, and API products. That management-model identity is distinct from the protocol data used to authorize an OAuth request. The authorization-code flow uses registered client details and user authorization, rather than a developer-name validation, when issuing and exchanging authorization codes. Google's Apigee documentation describes OAuth flows in terms of client applications, authorization servers, authorization codes, and access tokens. The underlying OAuth specification likewise defines the authorization-code exchange without a developer-name or device-ID validation operation. See Apigee OAuth 2.0 overview and RFC 6749, Authorization Code Grant.