MuleSoft®

WHITEPAPER

Principles of microservice security

How to manage digital risk at scale

Contents

Executive summary

Challenges

- Organizations are increasingly adopting microservices architectures to create a foundation for continuous innovation and agility, but the financial incentives associated with increased agility are tempered by the fear of exposing the valuable information microservices process. By 2020, Gartner predicts 60% of digital businesses will suffer major service failures due to the inability of the IT security team to manage risks.
- The frameworks, protocols, and technologies that have evolved over the years to solve security problems are multiple, complex, and sometimes rather difficult to understand. Choosing the right ones, applying and combining them appropriately, and leveraging them successfully is not an easy task.
- These difficulties will be increased in a microservice architecture because its many moving parts necessarily multiply the number of required security checkpoints.

Recommendations

- Understand and establish a “CIA” strategy for your microservice architecture: how can you ensure confidentiality, integrity, and availability of the information that they process?
- Apply a unified set of security principles and recommendations that can ease how you design and implement microservices that are secure at scale.
- Consider deploying a unified platform that will allow you to govern the entire lifecycle of microservices as they mature from design to implementation to management and finally to deprecation and retirement.

Introduction

Microservices do two things. They respond to queries for information like “Give me a summary of all the encounters realized by any clinician with this patient over the last 6 months.” And they execute commands like “Book me a flight to New York on the morning of March 16th.”

In a microservice architecture, a query or command will, by the very nature of the architecture’s functional decomposition, result in chains of API calls and messages across multiple microservice instances. Regardless of where a particular microservice sits in the chain, we must not lose sight of the user at the frontend.

This is because the user must be the focus of our security mechanisms. We are ultimately either enabling the user or rejecting her request. So in each of the above two scenarios, we must ask the question: who is “me”? The security mechanisms must identify this person and propagate her identity through every call. The identity of the client, namely the user interface and each microservice that participates in the query or command initiated by the user, is also of interest.

The security mechanisms must authenticate users and clients by verifying their identity claims in every invocation. Subsequently, they must make access control decisions about whether both the user and the client are authorized to execute the microservice.

Also, a security strategy critically must go beyond mere authentication and authorization mechanisms for any particular user and attend to the problem holistically. It must counterbalance the general goals of the microservice architecture to expose business capabilities across every user channel by placing restrictions on what or how much is exposed on any one channel. It must guide the business in meeting the goals of information security.

The three goals of information security

A microservice architecture is deemed secure when, guided by the seven design principles of microservice security, the goals of information security are met. These goals address both queries for information and execution of commands to ensure:

1. **Confidentiality:** The information they process is hidden from all users except those users who have been explicitly authorized to see it.
2. **Integrity:** The information they process is trustworthy because it is guaranteed to be correct and free from manipulation, malicious or otherwise, by a third-party.
3. **Availability:** The information they expose in queries is available every time it is needed by authorized users and the commands they execute are reliably brought to completion.

The goals of information security can be most effectively met through the application of security design principles. In this whitepaper, we use security design principles to guide you in the design of the microservices, how they are implemented, and how and where they are deployed. Given that most of the complexity of microservice security exists within the multiplicity of frameworks, protocols, and standards that abound, we avoid prolonged discussion about potential solutions and are very prescriptive in our recommendations.

An overview of the seven design principles of microservice security

The principles of microservice security should be pervasive in the design, implementation, deployment, and management of microservices to realize the above goals.

1. **Standardized:** Microservices should be protected with industry standard frameworks and protocols.
2. **Targeted:** Microservices should cater only to the specific capabilities needed by the users that consume them.
3. **Minimized:** Microservices should process the minimum amount of business information considered sufficient for the user's request.
4. **Locked:** Microservices should not be accessible by default, and they should require security credentials from the client upon every invocation.
5. **Multi-keyed:** Microservices should be accessible only if the client presents at least two security credentials.
6. **Elastic:** Microservices should execute in highly available environments dynamically growing and shrinking their instance count to meet traffic demands.
7. **Resilient:** Microservices should execute their commands guaranteeing that even when errors occur their work will be completed eventually.

How the seven principles manifest in an API strategy

A modern architecture that aims to facilitate IT agility and responsiveness to business requirements should establish core digital building blocks with well-defined APIs. These represent the most strategically advantageous means to access your data and connect digital touchpoints with backend systems of record. API-led connectivity aims to create reusable assets which will deliver value to the business in multiple contexts.

System APIs are designed with enough abstraction to hide the underlying systems with which they integrate. They are primarily encapsulations of capabilities centered around business entities, like Customer, Order, Invoice, and they are agnostic to business processes. They are driven by direct API calls and indirectly by their subscription to domain events. **[minimized, resilient]**

Process APIs are abstractions around business processes and complex interactions with users. They compose and reuse system APIs through explicit API orchestration (direct calls) and the more resilient API choreography using domain events. **[targeted, resilient]**

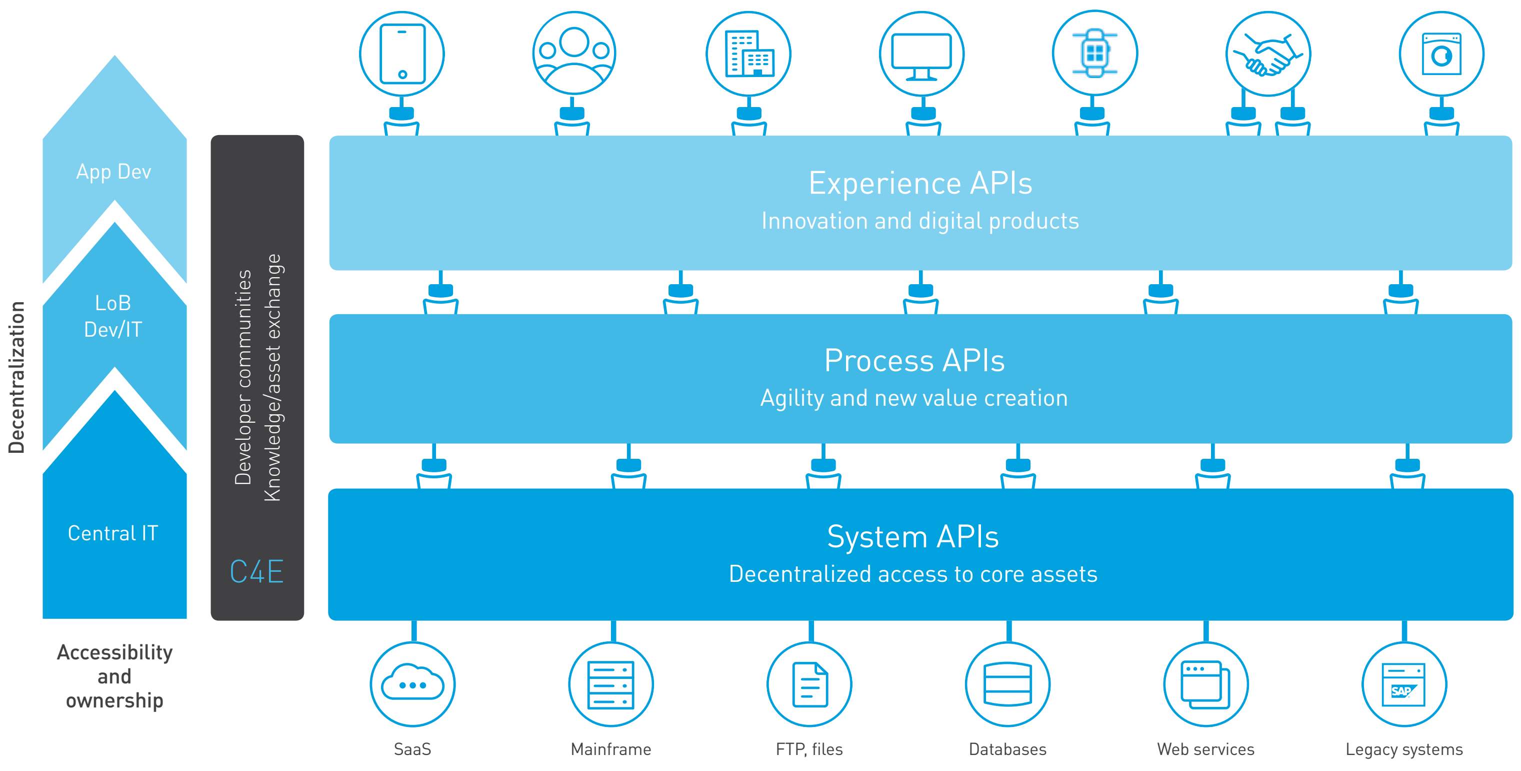


Figure 1: API-led connectivity

Both system and process APIs digitize business capabilities and form discrete digital building blocks that are individually scalable for use in initiatives that engage with users across multiple channels. They should not be invoked directly by any client software but through adaptations specific to the business channels that need them. **[targeted, locked]**

Experience APIs are adaptations of the core capabilities encapsulated in Process and System APIs. They are targeted at the specific requirements of the business channel and touchpoint that consumes them. A separate Experience API is delivered to each channel. They expose only the capabilities required for their channel. **[targeted, minimized]**

All three layers are protected with mechanisms that identify clients and users using industry standards to authenticate and authorize the callers**. [locked, multi-keyed, standardized]**

All three layers hide failures from the user when appropriate and their APIs are individually scalable. **[elastic, resilient]**

Notation used in this Whitepaper

The keywords “MUST”, “MUST NOT,” “REQUIRED,” “SHALL,” “SHALL NOT,” “SHOULD,” “SHOULD NOT,” “RECOMMENDED,” “NOT RECOMMENDED,” “MAY,” and “OPTIONAL” in capitals in this document are to be interpreted as described in RFC 2119.

The principles of microservice security

We present the principles to you as a coherent unified set. They should not be applied in isolation. Each of them relates to at least one other. Standardized for example is pervasive in all of the others. Targeted and Minimized both aim to reduce the degree of capability sharing among groups of users. Locked and Multi-keyed seek explicit fail-safe authorization every time a microservice is invoked. Elastic and Resilient have a particular focus on guaranteeing the availability of the microservices. We stress the need for a deliberate application of all of the principles to fully secure your microservice architecture.

Standardized

Standards now abound in the industry to address information security goals. HTTPS uses TLS to cater to both the confidentiality and integrity of information. It encrypts all of the data transported and uses digital certificates and digital signatures to identify both microservice and client. OAuth 2.0 offers a powerful framework for token-based access control.

OpenID Connect sits on top of OAuth 2.0 offering even more with tokens that deal with the identity of the authenticated user. It also has the same capabilities as SAML 2.0 offering federated SSO interoperability. RSA 2048+, AES 256+, and Blowfish are excellent encryption standards. NIST and PCI-DSS are excellent tokenization algorithms. It is preferable to use public standards for security mechanisms as any flaws in these will be discovered much quicker, given the huge proliferation of their usage, than anything you can build yourself.

Use TLS on all APIs

Every microservice SHOULD be invocable through a REST API. The exposed API MUST be over HTTPS. Private and semi-public APIs (see Profile the APIs on your Microservices) MAY use self-signed certificates, but public APIs MUST NOT use self-signed certificates. They MUST use certificates signed by a certificate authority.

Use API Manager policies

It is RECOMMENDED that you delegate security logic to policies that you can apply to the microservices from API Manager. Where a particular policy is not available out of the box, you SHOULD implement the logic in a custom policy.

Use a federated identity provider

You SHOULD have a federated identity provider to cater for the mechanisms, like OpenID Connect, OAuth 2.0, and SAML, that we recommend in this whitepaper. Anypoint Platform integrates seamlessly with SAML 2.0 identity providers. You can manage the identity of end users, apps, and administrative users of Anypoint Platform when this integration is established. Your identity provider SHOULD be able to federate its capabilities with other providers to facilitate cross-organizational security decisions.

Use a tokenization service

You SHOULD choose tokenization as your data masking strategy. You MUST use a tokenization service that is external to the deployments of your microservices and databases. If you have your own datacenter deployment you MUST have it inside the corporate segment of your network. (See Segment Deployments along the lines of your Network Zones).

Use token-based access control

You MUST enforce clients to use token based credentials when they invoke your APIs. You MUST oblige the clients to pass the token as a header with each API call. You MUST NOT rely on username/password credential pairs given the low entropy (high predictability characteristic of these passwords).

Targeted

The scope of responsibility designed for a microservice should be focused on the groups of users that use their capabilities. Ultimately, it is human beings that our strategy aims to protect the microservices from. Malicious software like bots is programmed by human beings to expose confidential information and capabilities to unauthorized users. Thus it is preferable to target the microservices at particular groups of users and avoid sharing their capabilities among multiple user groups.

Segment deployments along the lines of your network zones

It is commonplace in network topologies to segregate parts of the network into zones that generically we can label as:

1. **Corporate zone:** Strictly private traffic.
2. **Hybrid zone:** Segmentation that allows CloudHub deployments access to limited microservice deployments and systems of record in your data center.
3. **DMZ:** Demilitarized zone for traffic originating from the Internet.
4. **Extranet:** Similar to the DMZ but for Experience APIs exposed to apps in the hands of employees outside your data center.

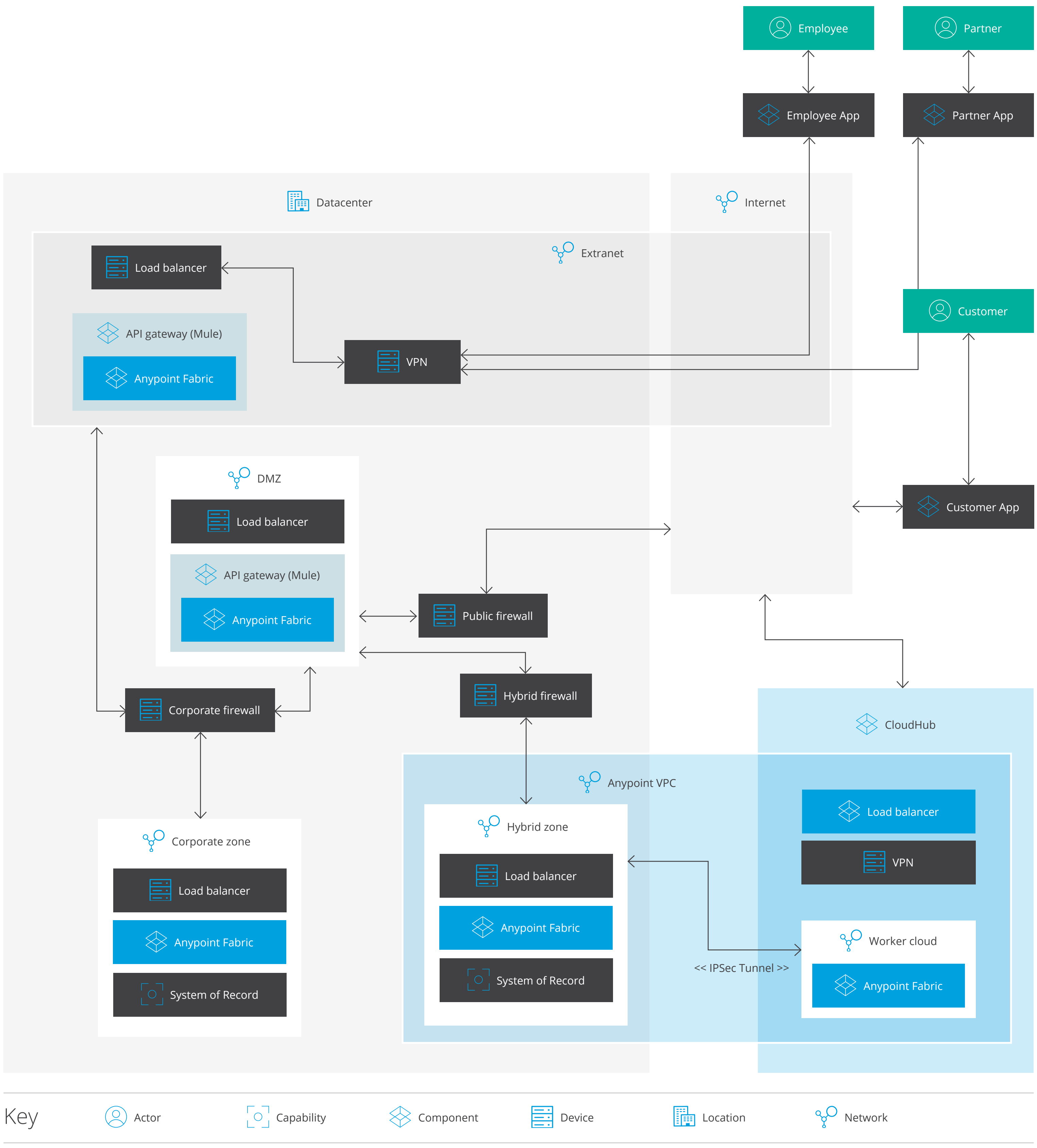


Figure 2: Microservice deployments in Network Zones

Network segmentation seeks to segregate traffic within and outside of your network so that strict rules can protect your microservices while facilitating necessary interaction with entities across the internet. Cross-segment API calls MUST be strictly controlled with firewall rules. All Internet originating API calls except for those originating from the extranet MUST pass through the DMZ. You MUST deploy an API gateway within the DMZ. You MUST NOT allow any inbound API calls from the Internet to the DMZ except for HTTPS over port 443. It is RECOMMENDED that no outbound connection is opened from within the DMZ with the exception of outbound calls from Mule runtime engine to API manager and runtime manager over

HTTPS and port 443. You MAY as a stricter alternative establish proxies to your customer Experience APIs in the DMZ and move the customer experience APIs to the corporate zone. You MUST NOT allow traffic directly to the corporate zone from the Internet or from the hybrid zone.

CloudHub deployments MUST NOT call microservices deployed to the corporate zone directly. They may call APIs in the hybrid zone. Microservices deployed to the hybrid zone MUST NOT call microservices deployed to the corporate zone. They SHOULD instead call gateway APIs deployed to the DMZ.

Profile the APIs on your microservices

The target network segment for your experience microservice deployment MUST be determined by the target user audience of the app that consumes it. You MUST profile your APIs according to their deployment zone to help you determine appropriate security measures for them.

		Target user audience		
		Customers	**Partners**	**Employees**
Network segment	Corporate	Private		Private
	DMZ	Public		
	Extranet		*Semi-public*	*Semi-public*

The above table profiles the Experience APIs exposed in the DMZ as public, those exposed in the extranet as semi-public, and those exposed in the corporate segment as private. We should use these API profiles when considering our choice of grant type with OpenID Connect and OAuth 2.0 (see Strategize on Choice of OAuth 2.0 Grant Types). Process and System APIs are necessarily profiled as private.

Set up an API gateway

You MUST deploy experience APIs that act as a gateway to all backend process and system APIs. These are the primary target for the application of security policies like OAuth, rate limiting, auditing, and data filtering. They SHOULD only compose process and system APIs which belong to their own domain. Domain events from other domains MAY also be subscribed to from these gateways unless they are public APIs deployed to the DMZ. Cross-domain API calls SHOULD be made through the target domain's gateway.

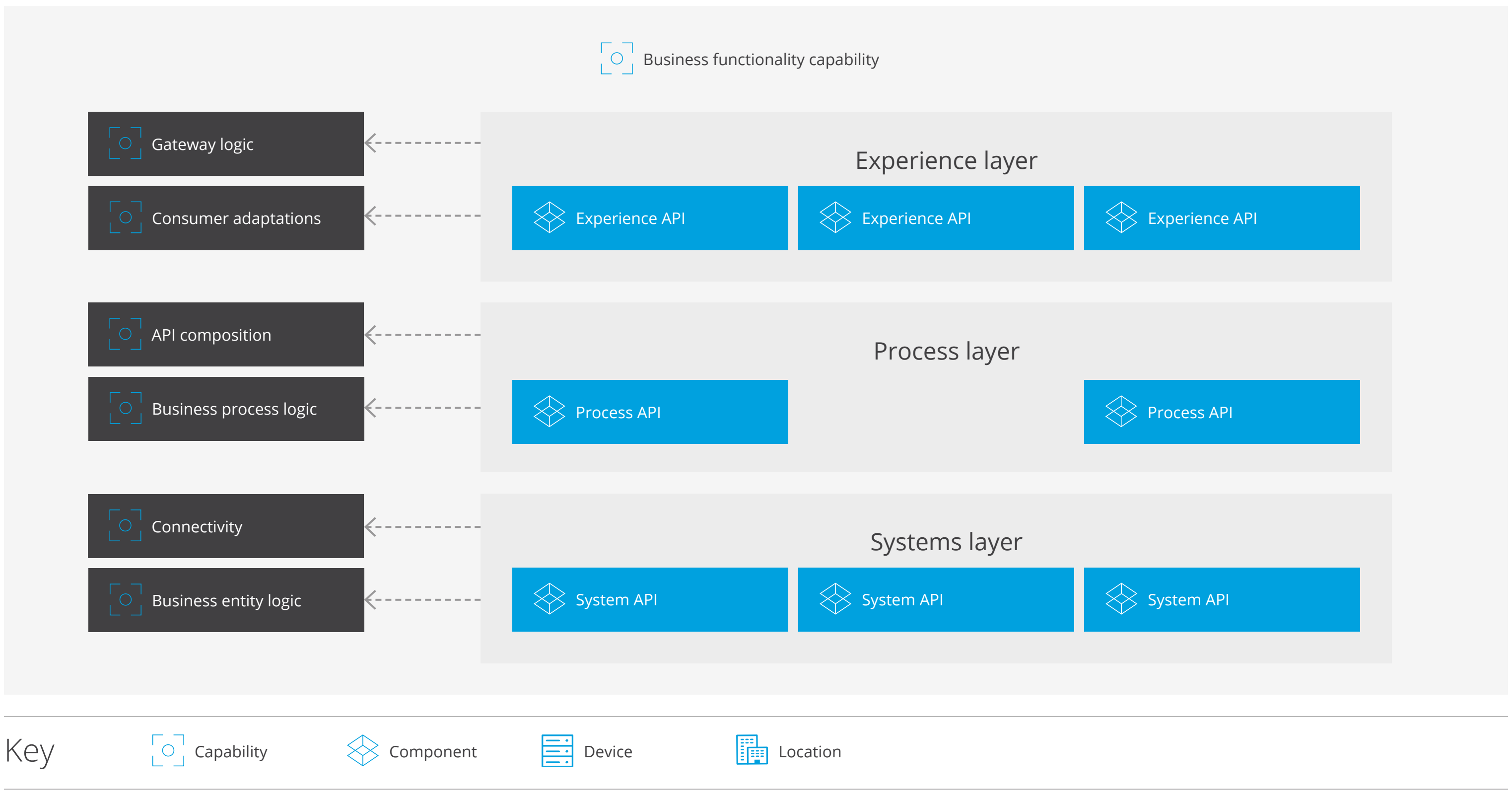


Figure 3: Microservice Classification and Gateway Logic

You SHOULD NOT allow any consuming app whether mobile, web-based, desktop, or otherwise to directly consume your Process and System APIs. It is often the case that these clients will have their own security constraints and other customization requirements. Given that many clients will need to consume one microservice, this could lead to a lot of refactoring as more and more clients come online. The adaptations we described in Experience APIs are the RECOMMENDED approach to take in order to expose the core business capabilities in your process and System APIs to the consuming channels.

Minimized

Microservices should not be generous with information. They should only respond to queries with information sufficient to meet the needs of the request and no more.

Design your microservices around business entities and processes:

The principles of domain-driven design identify domain aggregates as those business entities which are the root of a graph of entities and value objects that form a single transactional boundary. You SHOULD consider aggregates to be your primary candidate microservices. By decomposing the information sets you exchange with your stakeholders within the boundary of an aggregate, you allow each microservice to be consumed with the exact security requirements needed for its data. You can also independently scale the microservice in line with the demands for that particular business entity.

It is RECOMMENDED that you decompose the functionality in your architecture along the lines of business capabilities or complex interactions with your stakeholders. Often, many business entities must be utilized within the execution of a single command. Thus, microservices which encapsulate such capabilities are responsible for the composition of business entity microservices through API calls and domain events.

Model the design of your microservices within the context of business domains

It is RECOMMENDED that you use domain driven design which will help you limit the scope of responsibility of your microservices to a particular core domain of user activities within your organization. This is achieved by designing them within the bounded context established by the team of domain experts and engineers responsible for their design and development. The team models microservices using language that expresses a mutual understanding between the developers and the domain experts. Consequently, microservices which fall within those boundaries MAY have a particular perspective on a business entity, for example, which differs from the perspective of other teams working on other domains. You SHOULD limit the scope of your microservice responsibilities to the sub-domain for which they are built.

Filter data returned to client

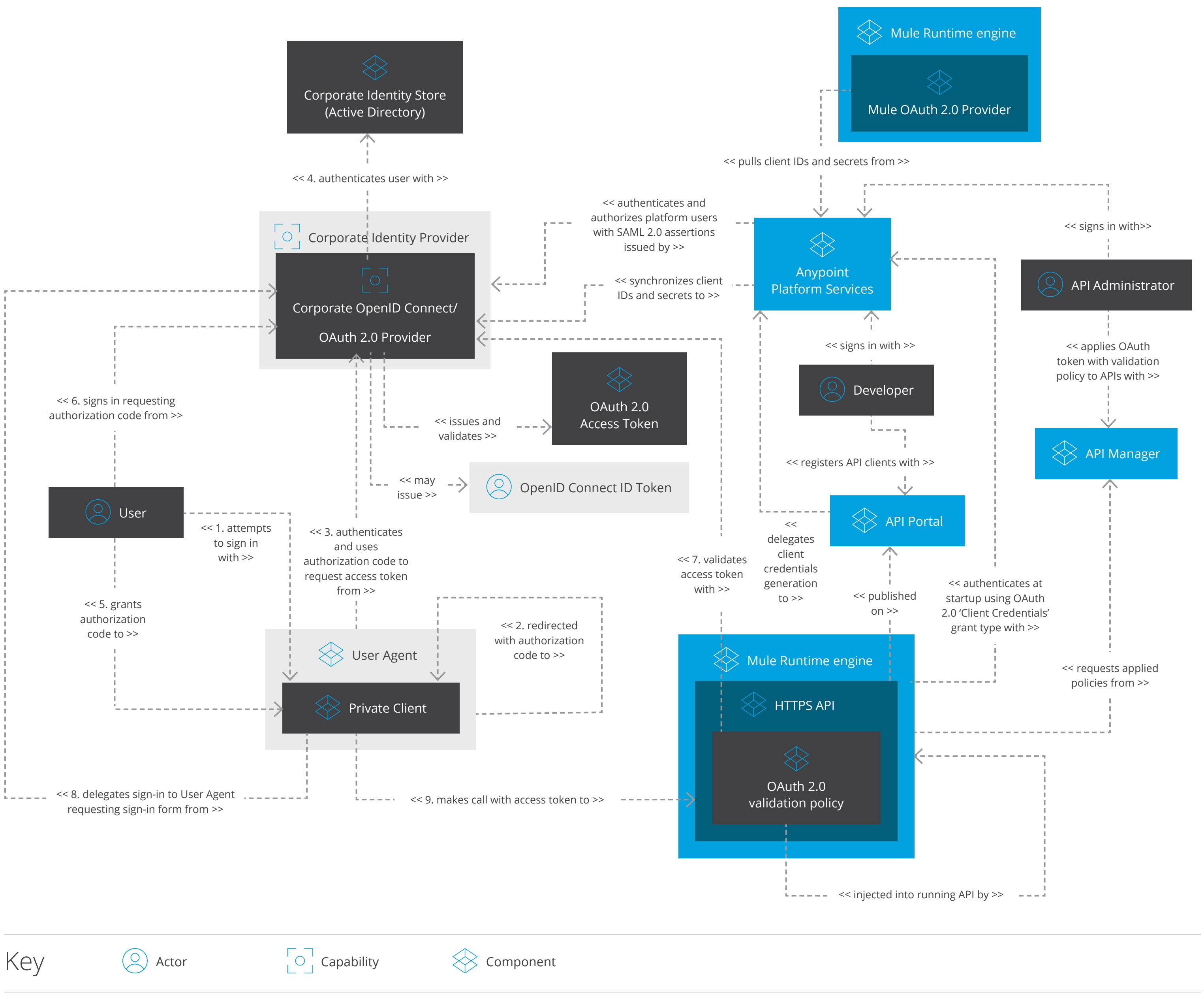
The full business capability encapsulated by process and system level microservices SHOULD be adapted through Experience APIs tailored to the specific needs of the consuming apps (see Security Design Principles and API-led Connectivity). These adaptations SHOULD act as facades to the microservice they consume and return only datasets appropriate to the consuming app and the users of the app. This is a static application of data filtering. To meet the more dynamic requirements of ABAC (see Use Attribute-based Access Control (ABAC)) you MUST be able to filter data on the fly. You SHOULD encapsulate this logic in a custom policy wrapped in an "<after />" element so that the data is filtered in the response flow. You SHOULD use DataWeave's dynamic transformation capability so that you can configure the transformation logic at policy application time.

Locked

Microservices must not be accessible for default. Every call must be authenticated and authorized to execute the query or command. Even public Experience APIs that expose information considered to be of the public domain must only be consumed by registered client applications. Private Process and System APIs should only ever be consumed by Experience APIs. The information microservices handle in queries and commands must not be visible to third parties and must be guaranteed to be correct.

Use OpenID Connect and OAuth 2.0

Use OAuth 2.0 and OpenID Connect to issue and validate bearer tokens to your API clients. You MUST oblige all consumers of Experience APIs to pass in an OpenID Connect ID token as well as the access token. The access token addresses client-oriented access control and the ID token addresses the REQUIRED user-oriented access control (See Apply User-Oriented Access Control).



Mule Runtime engine
Mule OAuth 2.0 Provider
Corporate Identity Store (Active Directory)
<< pulls client IDs and secrets from >>
<< 4. authenticates user with >>
<< authenticates and authorizes platform users with SAML 2.0 assertions issued by >>
<< signs in with>>
Corporate Identity Provider
Corporate OpenID Connect/ OAuth 2.0 Provider
<< synchronizes client IDs and secrets to >>
Anypoint Platform Services
API Administrator
<< signs in with >>
<< applies OAuth token with validation policy to APIs with >>
Developer
<< 6. signs in requesting authorization code from >>
<< issues and validates >>
OAuth 2.0 Access Token
API Manager
<< registers API clients with >>
<< may issue >>
OpenID Connect ID Token
User
API Portal
<< 1. attempts to sign in with >>
<< 3. authenticates and uses authorization code to request access token from >>
<< delegates client credentials generation to >>
<< 7. validates access token with >>
<< published on >>
<< authenticates at startup using OAuth 2.0 'Client Credentials' grant type with >>
<< 5. grants authorization code to >>
<< 2. redirected with authorization code to >>
User Agent
Mule Runtime engine
<< requests applied policies from >>
Private Client
HTTPS API
OAuth 2.0 validation policy
<< 8. delegates sign-in to User Agent requesting sign-in form from >>
<< 9. makes call with access token to >>
<< injected into running API by >>
Key
Actor
Capability
Component

Figure 4: Identity Provider and OAuth 2.0 / OpenID Connect with Anypoint Platform

Strategize on Choice of OAuth 2.0 Grant Types

There are four basic grant types to choose from in OAuth 2.0 but you MUST NOT use the 'resource owner password credentials' grant type as it goes against the best practice token based authorization OAuth 2.0 promotes. Also, the 'client credentials' grant type is only viable for and is RECOMMENDED for use in protecting System and Process APIs (see Protect Process and System APIs with OAuth 2.0 Client Credentials). Thus you MUST only use either 'implicit' or 'authorization code' when looking at your options for Experience APIs.

OAuth 2.0 classifies clients as private (those that can keep their credentials a secret) and public (those that cannot keep their credentials a secret). According to this classification mobile apps and Javascript single page apps are necessarily public clients. This will help us determine the appropriate grant type to use. 'Authorization code' was designed for server-side software clients like the older style JSP web apps. These can safely hide their credentials. Hence, clients in the authorization code scenario MUST authenticate. Client authentication is not mandated in the 'implicit' grant type.

		Grant type		
API profile	**Tier**	**Authorization code**	**Implicit**	**Client credentials**
Public	Experience		✓	
Private	Experience	✓		✓
	System/ Process			✓
Semi-public	Experience	✓		

While OAuth 2.0 prefers the use of 'authorization code' when possible and highlights certain weaknesses of 'implicit', you must be aware of the constraints that an OAuth 2.0 public client places on the choice of grant type. 'Authorization code' is certainly the stronger of the two grant types. However, you need to weigh up the impact of the exposure of a client secret to a malicious third-party against the exposure of an access token (which the 'implicit' grant type allows) to the same entity).

Hence, those APIs which you have profiled as public MUST use the 'implicit' grant type. Given that semi-public APIs are protected by a VPN we can rest assured that the exposure of client id and secret will be of no avail with the caveat that the identity provider instance exposed to semi-public clients must be different to the one exposed to public clients.

Protect public Experience APIs with OpenID Connect implicit flow

Public Experience APIs, which are those exposed to mobile apps and Javascript apps in the hands of your customers across the internet, MUST be protected with OpenID Connect Implicit Flow. The app is not obliged to authenticate itself. Your identity provider acting as OID provider will use its redirect URI to identify it. The consuming app SHOULD use multi-factor authentication to authenticate the user and it MUST validate the signature on the ID token that it receives at the end of the flow. The ID token is a JWT token and as such can be validated by validating its signature. The app MUST pass both the access token and the ID token in separate headers in the API call. You MUST apply an access token validation policy to the API. You MUST also apply an ID token validation policy which is able to verify the signature on the ID token and inspect the body for attributes that inform the access control decision (see Apply User-Oriented Access Control). All policy validation failures MUST reject the API call and return the HTTP status 403.

Protect semi-public employee APIs with OpenID Connect Authorization Code Flow

Semi-public employee Experience APIs, which are those exposed to mobile apps and Javascript apps in the hands of your employees across the extranet, MUST be protected with OpenID Connect Authorization Code Flow. The consuming app

MUST authenticate itself with your identity provider, it MUST use multi-factor authentication to authenticate the user, and it MUST validate the ID token that it receives at the end of the flow. It MUST pass both the access token and the ID token in separate headers in the API call. You MUST apply an access token validation policy to the API. You MUST also apply an ID token validation policy which is able to verify the signature on the ID token and inspect the body for attributes that inform the access control decision (see Apply User Oriented Access Control). All policy validation failures MUST reject the API call and return the HTTP status 403.

Protect semi-public partner APIs with JWT Tokens or SAML assertions and Client ID enforcement

Semi-public partner Experience APIs, which are those exposed to apps in the hands of your partners across the extranet, MUST be protected with either JWT tokens or SAML assertions. You MUST apply a custom policy to the API which will validate the signature in the token or assertion. You MUST apply the API Manager client ID enforcement policy to the client apps registered to consume these APIs.

Protect private Experience APIs with OpenID Connect Authorization Code Flow

Private Experience APIs, which are those exposed to server-side web apps in the hands of your employees within the corporate network, MUST be protected with OpenID Connect Authorization Code Flow. The consuming app MUST authenticate itself with your identity provider, it is RECOMMENDED that it use multi-factor authentication to authenticate the user, and it MUST validate the ID token that it receives at the end of the flow. It MUST pass both the access token and the ID token in separate headers in the API call. You MUST apply an access token validation policy to the API. You MUST also apply an ID token validation policy which is able to verify the signature on the ID token and inspect the body for attributes that inform the access control decision (see Apply User-Oriented Access Control). All policy validation failures MUST reject the API call and return the HTTP status 403.

Protect Process and System APIs with OAuth 2.0 Client Credentials

Process APIs, which are those that encapsulate your core business capabilities, MUST NOT be consumed directly by any consuming app. They MUST only be consumed by the experience microservices that adapt them. The experience microservice as client to the Process APIs MUST be registered with API Manager as an app and be given its own client ID and secret. It MUST store these encrypted either in the secure credentials vault or encrypted in a database. It MUST request an OAuth 2.0 access token from your identity provider using the 'client credentials' grant type. It may cache the access token for subsequent invocations. It MUST request a new access token before the previous one expires.

System APIs, which are those that encapsulate your core business entities, MUST NOT be consumed directly by any consuming app. They MUST only be consumed by the experience microservices that adapt them or process microservices that compose them. The experience/process microservice as client to the System APIs MUST be registered with API manager as an app and be given its own client id and secret. It MUST store these encrypted either in the secure credentials vault or encrypted in a database. It MUST request an OAuth 2.0 access token from your identity provider using the 'client credentials' grant type. It may cache the access token for subsequent invocations. It MUST request a new access token before the previous one expires.

Mask sensitive Data at rest

You MUST NOT persist sensitive personally identifiable information or security properties like access tokens, usernames, and passwords in plain text to any systems of record, or logs, or message brokers.

Use encryption

You MAY choose to encrypt the entire payload before persistence. In this case, you will need to decrypt it before sending it back to a user or before subsequent processing. When encrypting you MUST choose strong cryptographic algorithms like RSA 2048+, AES 256+, or Blowfish. For stronger security requirements you MUST use algorithms that have are compliant or certified with industry standards like FIPS 140-2.

Use tokenization

It is RECOMMENDED that you adopt tokenization in preference to encryption. You MUST tokenize sensitive data as soon as you receive it in its plain text values. Whenever the plain text value is needed for subsequent processing you MAY detokenize it.

You MUST NOT detokenize any tokenized values unless needed for processing or in response to an explicit user query for that information. You MUST respond to other user queries with tokenized values. You MUST ensure that the algorithm used is based on industry standards like NIST or PCI-DSS.

Mask sensitive data in transit

You MAY exploit the encryption of HTTPS to mask data in transit to Experience APIs. You MUST call Process and System APIs and publish domain events with sensitive data either tokenized or explicitly encrypted.

Multi-keyed

There should be no single point of failure in your security mechanisms. It is preferable to have multiple checks so that if one mechanism were to fail then the others will still protect the microservice. The client must present credentials to identify itself and the user who initiated the request.

Apply user-oriented access control

An inherent weakness to OAuth 2.0 (see Use OpenID Connect and OAuth 2.0) is the assumption that the user, on behalf of whom the access token was issued to the consuming app (client, is the owner of the resource). You MUST NOT allow access to your Experience APIs until you have verified the identity of the user including information about them which will inform the access control decision. You SHOULD use the OpenID Connect ID token to help with this decision. You SHOULD propagate the ID token across every API call made by all the microservices that participate in a user transaction.

Use Role-based Access Control (RBAC)

You MAY make access control decisions based on the group to which the user belongs. You SHOULD look for this group name in the OpenID Connect ID token. There is no provision for this in the standard but the standard is extensible and you SHOULD configure your identity provider to return the group name. You SHOULD NOT hard-code role names in the ID token as roles are specific to the microservices and are based on the knowledge of which group the user belongs.

Use Attribute-based Access Control (ABAC)

It is RECOMMENDED that you adopt the more dynamic policy based access control known as ABAC. User attributes such as group are still important but they are coupled with other dynamic attributes such as the time of day. You SHOULD externalize the policy decision point (PDP) to your identity provider and make an API policy the policy enforcement point).

Use Multi-factor Authentication on user apps

You MUST use multi-factor authentication with users at the frontend. This MAY take the form of simple 2-factor authentication.

Use mutual TLS on private and semi-public APIs

You MUST use mutual TLS on all semi-public experience API calls from within the extranet and on private API calls from the DMZ to the corporate segment. It is NOT REQUIRED to do the same on public API calls given the much larger customer base that consume those APIs.

Propagate OpenID Connect ID tokens through all private API calls

You SHOULD pass OpenID Connect ID tokens received in all Experience APIs to the process and System APIs that these consume. You SHOULD apply policies from API manager that validate these tokens and check for group information on them. It is the group information that will facilitate at least role-based access control.

Use mutual TLS on all system integrations

You MUST use mutual TLS when a microservice exchanges data with a system of record. JDBC, JMS, AMQP, and LDAP calls facilitate the use of TLS.

Elastic

To make information available at every time it is needed, your microservices must be able to handle spikes in traffic. You should protect your APIs from the very traffic that consumes them. Even friendly traffic should be considered a threat as it can overwhelm your microservices. A strategy is needed for your microservices to grow and shrink with the traffic.

Protect public Experience APIs from Denial-of-service attacks

Denial of Service (DoS) attacks try to overwhelm a service by volume of calls or size of messages. All gateway API calls coming into the DMZ from the internet MUST be rate limited and have their payloads analyzed for threats. It is RECOMMENDED that you address these requirements with rate limiting and XML/JSON threat protection policies that you can apply with API Manager.

Use containerization

You SHOULD use containerization to deliver your microservices (See Use Container Scheduling). Apart from making the realization of continuous delivery easy, containers also bring security benefits to a microservice by decreasing the extent of security risk present within its environment. You MUST NOT pull random images from the internet. You MUST use base images with vetted sources that can be easily validated and checked. You MUST scan your containers for common vulnerabilities and exposures (CVEs) just like any other software distribution. You SHOULD add base image release metadata to containers.

Use container scheduling

You SHOULD use a container scheduling solution, usually a core component in a PaaS, to facilitate scaling up and down the number of instances of your microservices. This will guarantee that the processing capacity of your microservices grow and shrink with the traffic running through them.

Use service lookup for API invocations

You MUST decouple the location of any particular instance from its invocation by a client. Strong coupling, on the contrary, will lead to availability failures. Every microservice invocation MUST be on the basis of a prior service lookup to an external service registry.

Resilient

Most commands and some queries issued by users will result in multiple microservice calls. Microservice architectures must be designed as if failures are guaranteed. Errors that occur should be hidden from users when possible and appropriate.

Implement the circuit breaker pattern

You must implement the microservice with logic that gracefully handles errors on the backend resulting from other microservice invocations or system of record integration failures. You SHOULD consider the circuit breaker pattern as template to this error handling. The idea is to mimic what happens in an electrical circuit breaker. It is an automatically operated electrical switch designed to protect an electrical circuit from damage caused by overcurrent/overload or short circuit. Its basic function is to interrupt current flow after protective relays detect a fault. Similarly, a microservice with a circuit breaker will temporarily avoid further backend invocations when exceptions occur and exploit a fallback function that can return an appropriate cached dataset during that period. It should concurrently check the health status of the backend service or system and resume normal implementation flow when it is back online. With this mechanism in place you will avoid unnecessary propagation of errors and maintain service availability.

Use a message broker for event-driven scenarios

Domain event-driven composition of microservices is to be favored above API call orchestration when multiple microservices participate in the execution of a command. You MUST use a message broker like Anypoint MQ to publish and subscribe to the domain events. If a participating microservice is not available when the command was issued, it will not matter as it will receive the event when it is back online. This results in eventual consistency of the data and final completion of the command even when errors occur.

Isolate subscriptions to message brokers

Microservices subscribed to Anypoint MQ queues or exchanges, or those that subscribe to AMQP or JMS message brokers MUST NOT do so from the DMZ. These subscriptions SHOULD be limited to microservices deployed to the corporate or hybrid zones or to Cloudhub. Microservices deployed to the DMZ MAY publish messages to the message broker.

Summary

Table of requirements and recommendations

Principle	Requirement	API profile	Level
Standardized	TLS (HTTPS)	All	Required
	API manager policies	All	Recommended
	Federated IdP	All	
	Tokenization service	All	Required
	Token-based access control	All	
Targeted	API profiling/network segments	All	Required
	API gateway	All	
Minimized	Design by business entities and processes	All	Required
	Domain-driven design	All	
	Data filtering with API policies	All	

Principle	Requirement	API profile	Level
Locked	Oauth 2.0 client credentials	Private (process and system)	Required
	OpenID connect implicit flow	Public	
	Open ID connect authorization code flow	Semi-public	
	JWT tokens or SAML 2.0 assertions	Semi-public (partner APIs)	
	Mask sensitive data with encryption or tokenization	All	
Multi-keyed	RBAC or ABAC	All	Required
	Mutual TLS	Semi-public	Required
		Private	
	Open ID connect ID token propagation	Private	Required
	Mutual TLS on system integrations		
	Client ID enforcement policy	Semi-public	Required
Elastic	Rate limiting	Public	Required
	Containerization	All	Required
	Container scheduling	All	
	Service lookup	All	
Resilient	Circuit breaker pattern	All	Required

Glossary

Availability: Characteristic that information is always processable by your microservices.

Bounded Context: Linguistic boundary of mutual understanding between domain experts and developers of the purpose and scope of responsibility of a microservice.

Confidentiality: Characteristic that guarantees information is only visible to authenticated and authorized users.

Demilitarized Zone (DMZ): Network segment or subnetwork that adds a layer of security by restricting incoming traffic from the internet to only those microservices deployed to the DMZ.

Domain Driven Design (DDD): Set of software design principles that attempts to align software with specific domain understanding and protect it from differing viewpoints in other domains.

Domain Event: Encapsulation of a business domain specific transaction, which is published by a microservice to asynchronously and reliably invoke other collaborating microservices.

Encryption: The process of encoding data in such a way that only authorized users can see it.

Extranet/Virtual Private Network (VPN): Means to extend a LAN out across the internet while maintaining privacy and security.

Federated Identity Provider: A dedicated system that manages the identities of users, and apps, facilitates authentication, the issuing and validation of authorization tokens, acts as an OpenID Connect/OAuth 2.0 provider, and a Secure Token Service (STS) provider.

Integrity: Characteristic that guarantees information is not modified by a malicious third-party.

Json Web Token (JWT): Interoperable identity token used for authentication scenarios and incorporated into OpenID Connect flows.

Mutual TLS: Protocol that obliges the client to send a digital certificate to authenticate itself to the microservice. One way TLS consists in only the microservice authenticating itself to the client.

Network Segment: A division of a network that provides isolation accessible only when firewall rules are set up to allow access.

OAuth 2.0: A framework for the delegated authorization by authenticated users to apps of resources that they own.

OpenID Connect: Layer on top of OAuth 2.0 that addresses interoperability, user identity and user authentication.

Private API: Interface to microservice only accessible from within your LAN deployed to the corporate segment.

Public API: Interface to microservice deployed to your DMZ and accessible from the internet.

SAML 2.0 Assertion: Interoperable identity token used for authentication scenarios and incorporated into the WS security standard as well as single-sign-on (SSO) configurations.

Semi-public API: Interface to microservice accessible from the internet so long as a VPN connection is established.

Tokenization: the process of substituting sensitive data with a meaningless token.

Transport Level Security (TLS): Is a network communication protocol that aims to address the needs for information confidentiality and integrity during the exchange between two machines and does the same for microservice invocation.

Additional resources

We presented this whitepaper referring briefly to the goals of information security. We refer you to our companion whitepaper Protect your APIs: an in-depth analysis of general security concerns, the industry standards that have emerged to address those concerns, and the security capabilities of Anypoint Platform.

Likewise, we explore microservices only from the security perspective. We refer you to Best practices for microservices: an in-depth analysis of the business and technical drivers behind a microservice architecture, the principles involved in designing microservices, as well as capabilities on Anypoint Platform that facilitate a microservices implementation.

About MuleSoft

MuleSoft's mission is to help organizations change and innovate faster by making it easy to connect the world's applications, data and devices. With its API-led approach to connectivity, MuleSoft's market-leading Anypoint Platform™ is enabling over 1,400 organizations in approximately 60 countries to build application networks.

For more information, visit **mulesoft.com**