



Quick Reference Card

— API —

Architecture stra+egy

There is a better way



API Architecture Strategy.

As soon as we start working on an API, architecture issues arise. Many mistaken common beliefs turn out to be fiction in this area. A poorly designed API architecture will lead to misuse or – even worse – not be used at all by its intended clients: application developers.

To facilitate and accelerate design and development of your APIs, we share our vision and beliefs with you in this Reference Card. They come from our direct experience on API projects.

API Levels.

The API level you are targeting can be reflected by the type of consumers you are addressing.



The main difference lies in the way you need to “industrialize” the enrolment process and the quality required for your API.

› At Level 1, if an application developer faces an issue, he can directly meet the guys from the API team

You should target Open API from the beginning (even if you are targeting Level 1 or 2 in the short term):

- › So that you can fully industrialize the way developers consume your “services” on your developer portal: <https://developers.fakecompany.com/>
- › This is the only way to offer good enrolment, TTFAC* & support in a digital way



Which API Strategy?



Integrating your legacy SOA implementations into your API Strategy... could end up with an **URBANIZATION Strategy**

Monitoring, Accounting.



Focusing on the REST approach inspired by Web Giants... may end up building a **state of the Art API**

RESTful, Developer portal, TTFAC* & DX**, X-device / X-channel.

* "Time To First API Call" is the time a developer needs to consume the API in production after reading the documentation on the developer portal! We target 5 minutes.

** "Developer experience". APIs are used by humans. We target a massive adoption, so we should craft them with love.



Why an API strategy ?

“Anytime, Anywhere, Any device” are the key problems of digitalisation. API is the answer to “Business Agility” as it allows to build rapidly new GUI for upcoming devices.

An API layer enables

- › Cross device
- › Cross channel
- › 360° customer view

Open API allows

- › To outsource innovation
- › To create new business models

Embrace WOA

“Web Oriented Architecture”

- › Build a fast, scalable & secured REST API
- › Based on: REST, HATEOAS, Stateless decoupled μ -services, Asynchronous patterns, OAuth2 and OpenID Connect protocols
- › Leverage the power of your existing web infrastructure

DISCLAIMER

This Reference Card does not claim to be 100% accurate. The design concepts shown here are a result of our previous work in the REST area. Please check out our blog <http://blog.octo.com>, and feel free to comment or challenge this API cookbook. We're really looking forward to talking with you.



API Big picture Architecture: From SOA to WOA.

SOA (Service Oriented Architecture) fulfilled the promise of breaking down the monoliths but its implementation came with many pitfalls.

WOA (Web Oriented Architecture) is a subset of SOA based on RESTful microservices and tends to correct mistakes from past implementations.



SOA*

SOA shouldn't be confused with its past implementations. From 2000, the implementation is WS-/SOAP at 99.%

ARCHITECTURE

Service oriented by design:

- multiple services can be provided,
- results in a huge number of services with slight variations depending on consumer needs.

.....
RPC approach: abstract of the distributed nature of the WWW network.

MAJOR IMPLEMENTATIONS & PROTOCOLS

Heavyweight specific protocols (WS*).

.....
Use custom applicative protocols on top of HTTP and SOAP, that developers have to learn before calling any service.

.....
Security is based on a VPN or/and complex WS standard designed for server to server exchanges.

INTEGRATION PATTERNS

Often implemented through a façade pattern which consists of a monolithic bloc.

.....
Often concentrates complexity in the hands of a centralized ESB tool/team which has no ownership of business referentials.

.....
Often thinking by vendor & tools first: ESB, BPM, BAM...

CONSUMERS

Clients are mostly servers.



WOA

ARCHITECTURE

Resource-oriented by design: ensures a unique representation of each resource, regardless of the number of consumers.

.....
Explicitly assumes the nature of the WWW network: strengths and weaknesses.

MAJOR IMPLEMENTATIONS & PROTOCOLS

Simple and based on common Web standards (HTTP, URI, DNS...) used since the 90s.

.....
Use HTTP as the “universal” applicative protocol. No need to reinvent the wheel so that developers can quickly use APIs, which offer good affordance.

.....
Security is based on simple Web protocols, device-agnostic by design: OAuth2 and OpenID Connect.

INTEGRATION PATTERNS

Suits both a façade pattern and a microservices pattern (the last one being the true distributed nature of the Web).

.....
In a microservices implementation, each API team is fully responsible for their product: each team is responsible for the quality (SLA) of their resources instead of putting it into an ESB.

.....
Think API First.

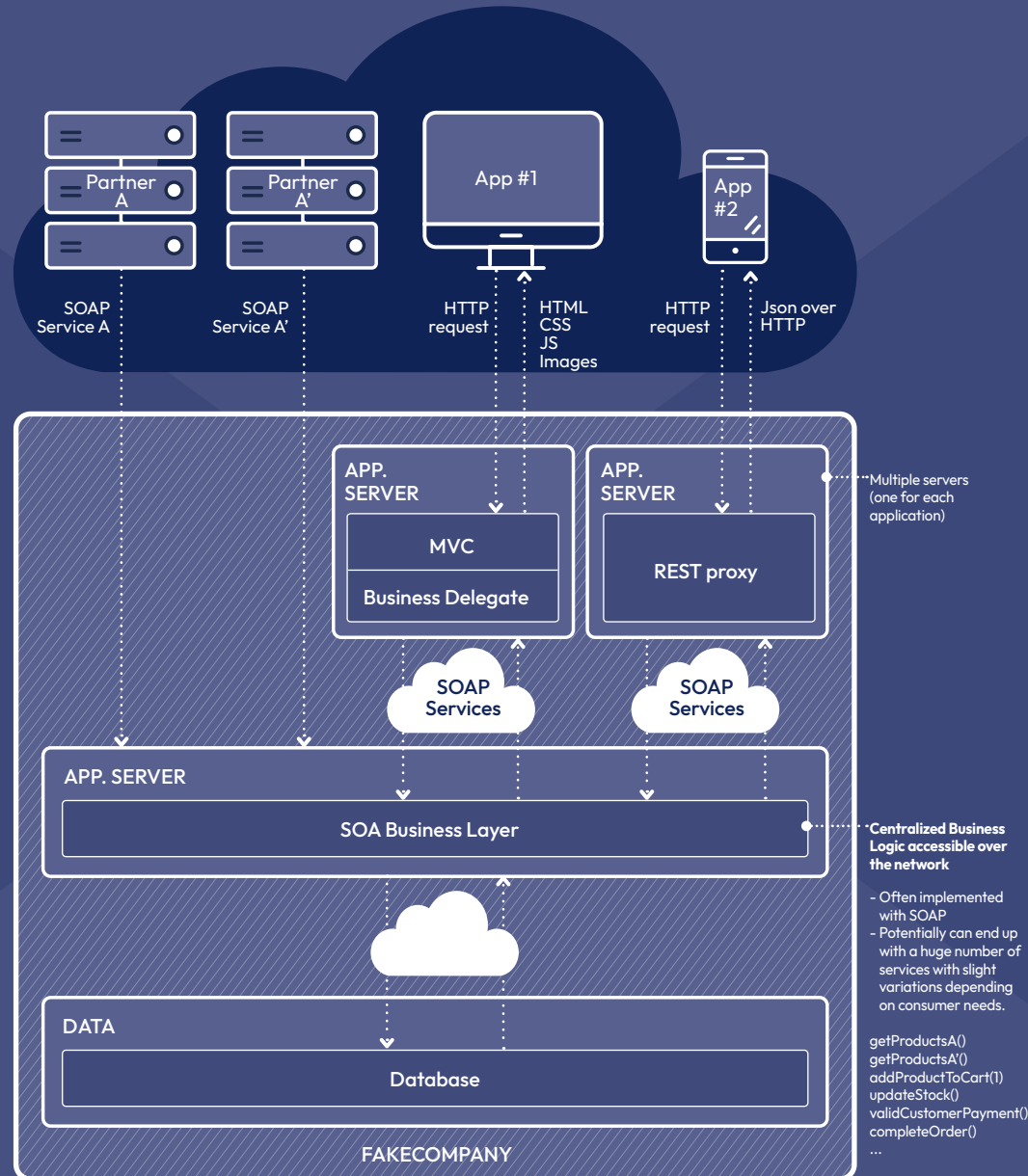
CONSUMERS

X-devices: clients are servers, native mobile apps, browsers...



SOA

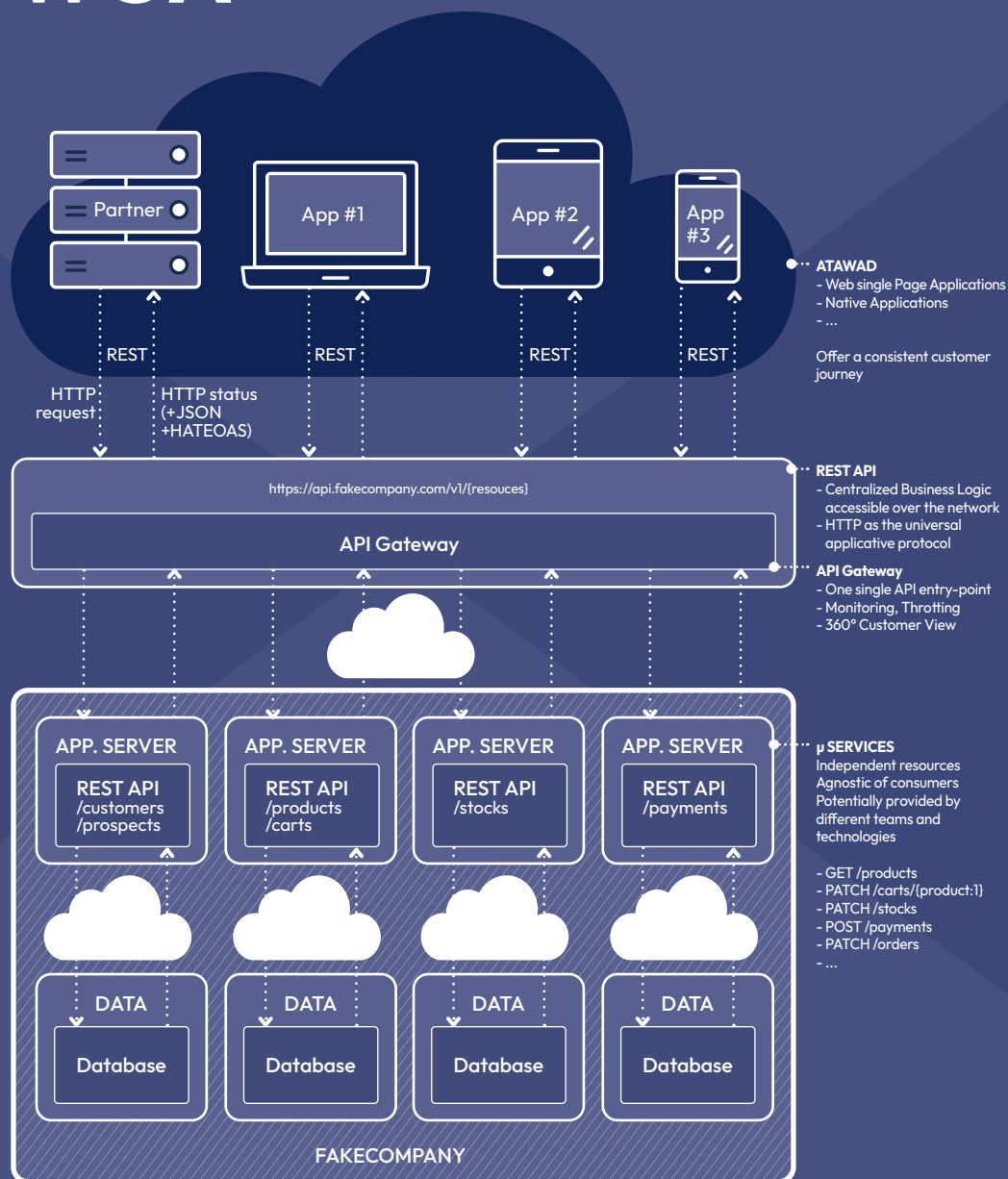
SERVICE ORIENTED ARCHITECTURE





WOA

WEB ORIENTED ARCHITECTURE
(REST API based &  services based)



SOAP vs REST.



INTERVIEW

Nick Gall

Portfolio Design Lead at IBM / VP Gartner at Group

“WS-* style Web Services are «Web» in name only... The W3C should extricate itself from further direct work on SOAP, WDSL, or any other WS-* specifications”

2007 - <https://www.w3.org/2007/01/wos-papers/gall>

David Orchard

Principal Engineer at @WalmartLabs / WS standards at BEA

“Given the complexity of just SOAP and WSDL, how many developers will really be able to move to the full stack?... The promise of WSDL 2.0 has not materialized and is unlikely to do so”

2007 - <https://www.w3.org/2007/01/wos-papers/bea>

Steve Loughran

Apache Axis committer

“The only place SOAP survives is in the enterprise because if you can control both ends of the conversation, you can use the same toolkit and eliminate interop”

2007 - <http://www.1060.org/blogxter/>

Antoine Chantalou

Head of WOA & API at OCTO Technology

“By choosing REST and Web Oriented Architecture, you’re putting all the chances on your side to succeed in your API strategy... SOAP is an amazing example of how businesses can embrace complex architectures and solutions”

2015 - APIdays conference in Paris



SOAP VS REST: IT'S ABOUT ARCHITECTURE *Style*

RPC & SOAP

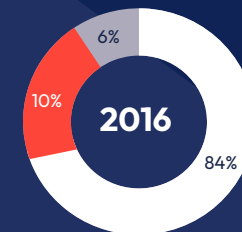
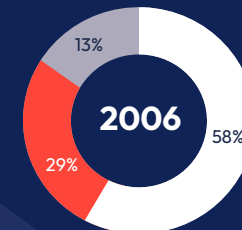
- › Are operation/service oriented
- › Tend to unify local and remote computation
- › Are contract & server oriented

REST

- › Is resource oriented
- › Explicitly uses WEB distributed architecture
- › Is developer oriented

Simplicity wins again

**SOAP is Not Dead -
It's a Zombie in the Enterprise.
Your API should be REST based**



- REST
- SOAP
- Other

Distribution of API protocols and styles, based on directory of APIs listed at ProgrammableWeb, May 2016.



API architecture stakes.

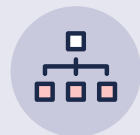


STATELESS

You should build stateless resource providers. A stateless API provider is easier to distribute, scale and cache.

Stateless doesn't mean that there is no state. There are actually two kinds of state:

- Client handles application/session state: where you are in the interaction, navigation, session.
- Server handles only resource state and no session. Each request is self-contained.



MICROSERVICES

Microservices is a key feature of Web Oriented Architectures. At some point, you may consider splitting your CORE IT systems into decoupled medium-grained microservices.

- Medium grained microservices would be: each resource provider is responsible for a set of resources that are functionally related
- Fine grained microservices would be: each resource provider is responsible for one resource

Business & technological stakes are:

- Short TTM & adaptability
- Keep complexity under control
 - Small and manageable resource providers
 - Limited component lifecycle synchronization
- Technology leveraging
 - Scalability & performance
 - Appropriate technology for the right usage

Note: if you decide to implement microservices, then this requires a compatible organization and culture (small short-cycle teams and Devops).



ASYNCHRONOUS

A request should always offer low latency. An acceptable response time could be 200ms.

1. On a functional level:

You should switch to an asynchronous flow if response times exceed the limit you consider acceptable. This limit could be 1s. Technically, the resource provider queues the request and sends 202 response to the client.

```
PATCH /orders/007 -d '{"status":"complete"}'  
< 202 Accepted
```

When the request is processed, the user is informed by another channel: HTTP2 server push, websocket, email, sms, or polling.

Asynchronous processing may have heavy impacts on business workflows.

2. On a technical level:

Requests sent by the client should always be asynchronous (XHR for browser's app).

The resource provider may be asynchronous (see non-blocking/reactive architectures). You will get a system that is concurrent by design.





NON-TRANSACTIONAL

With REST, you'll have to drop transactions (from a client point of view). HTTP is not transactional.

You may manage transactions with one the following solutions:

1. Use compensating actions by checking resource state, or request response, and implementing rollback logic inside API consumer.

For example, to book an hotel room and a plane ticket on two different providers:

```
POST https://api.hotel.com/orders
< 201 Created
< Location: https://api.hotel.com/orders/768
POST https://api.travel.com/orders
< 500
DELETE https://api.hotel.com/orders/768
```

An asynchronous batch could have also triggered an email to the user: "Sorry, your order could not be completed..." and revert to previous successful operations.

2. Adopt an explicit resource design which reveals the underlying transaction.

For example, to make a money transfer between resource A and B, instead of calling:

```
PATCH /accounts/a {amount : balance-XX}
PATCH /accounts/b {amount : balance+XX}
```

You could design a «transfer» resource that would allow:

```
POST /transfers {acc1 : a, acc2 : b, amount : XX}
```

Please note that ACID transactions are absolutely not mandatory. A good illustration to keep in mind is that "Starbucks is not transactional!"

You order, then you pay, then you get your drink. If something goes wrong between the two operations (which probably happens in 0.01% of all purchases) you'll deal with a compensating action.

You should not design an architecture based on exceptional behaviors.



INFRASTRUCTURE

You may consider Cloud hosting over OnPremise hosting, as API should be as close as possible to users' devices to provide an optimum response time:

› The "OAuth2 dance" to authenticate and authorize users involves several HTTP calls.

› Safe method calls should be cached with standard Web tools.

A WAF RP should not be used. The following issues should be addressed by your API platform:

› DOS/DDOS protection is generally offered by cloud IaaS, PaaS platforms.

› Authentication, Authorization, Accounting is provided by OAuth2/OIDC.

› Confidentiality, Integrity is provided by TLS.

› An RP that forbids any HTTP methods (PUT, PATCH, DELETE...) is incompatible with an API strategy.

API security.



HTTPS

All requests (OAuth2, IDP, API) must be secured with TLS (RFC5246).

- TLS provides confidentiality via encryption.
- TLS provides data integrity via keyed MAC (SHA-256).



AUTHORIZATION

API_KEY should be used to authorize **client** (application) on public resources
OAuth2 (RFC6749) should be used to authorize both **client** (application) and **users** on private API resources.

OAuth 1.0 (RFC5849) is now obsolete as OAuth2 brought two major improvements:

- Digital signature of requests were replaced by TLS.
- OAuth2 supports any kind of device (native, web browser, servers...).

Keep in mind that OAuth2 is mandatory to protect private resources (e.g. when the end user needs to be authenticated with a login screen):

```
...
/customer/007?client_id=API_KEY -H
'Authorization:Bearer X.Y.Z'
/customer/007/accounts?client_id=API_KEY -H
'Authorization:Bearer X.Y.Z'
```

OAuth2 is not mandatory for public resources. Client Credentials flow can be used if you wish to protect all your endpoints with OAuth2. But an API_KEY is sufficient for public resources:

```
/products?client_id=API_KEY
```



AUTHENTICATION

OpenID Connect protocol could be used to authenticate both **client** (application) and **user** on private API resources.

Once in place, OpenID Connect allows you:

- to manage your own OAuth2 instance to authorize access to API resources,
- to use your own OpenID Connect provider to authenticate users,
- to use any external OpenID Connect provider to authenticate users: Google, Facebook and so on...

Web Giants will probably handle authentication better than you ever can: two-factor authentication with sms...



COMMON PITFALLS

- If you are targeting “Digital”, you should not consider any other protocol: OAuth1, SAML2, etc. “OpenID Connect was designed to also support native apps and mobile applications, whereas SAML was designed only for Web-based applications”.
- Do not use encryption/signatures on the applicative side.
- Do not implement custom security solutions.
- A handy way of avoiding implementing your own OAuth2 provider is to use an out-of-the-box implementation.

API team organization.

Your API strategy will impact your teams and organization

As Conway's Law says: "Any organization that designs a system [...] will inevitably produce a design whose structure is a copy of the organization's communication structure".

- You should organize your teams as you would like your IT system to be.
- You should consider a small autonomous and empowered agile team to build your API. Let's call it the API squad.

stage

1

API MVP

When you are building your API, you generally create an API squad to set up some CORE functionalities that need to be centralized:

- API Management portal.
- Developer portal.
- OAuth2 or OpenID Connect.

This team could set up the first set of API resources to validate the API MVP, with a façade pattern, and that's OK.

The API squad should include all members that are responsible for it, typically:

- A Product Owner from Business or Marketing.
- Developers (optionally a Technical Leader).
- OPS.
- When it make sense, a Community Manager.

The total team members should not exceed 10.

API SQUAD
STAGE ONE

stage

2

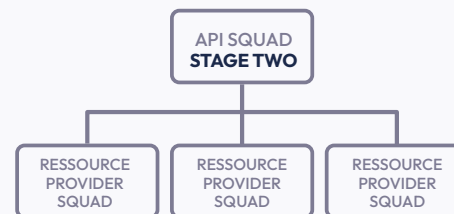
SCALING API

When the MVP is validated, you should switch to a microservices pattern (see "API integration patterns").

In order to scale, the first API team remains and continues to be responsible for the global API:

- The Product Owner ensures that the API is consistent.
- The role of the Community Manager increases.

Other resource provider teams have to be set up as needed using the same model. The API team "delegates" and coaches resource provider squads (REST, API design...). Each CORE IT component publishes REST resources that it is accountable for. API façade is progressively decommissioned.





API SQUAD



BUSINESS ANALYSTS [Business]

- Co-design API resources
- Write automated functional tests (TDR)



PRODUCT OWNER [Business]

- Sync development with other teams
- Responsible for API success
- Define Follow-up indicators



DEVELOPERS [IT]

- Design & develop API resources
- Write API documentation
- Measure and improve API performance
- Write unit automated test



COMMUNITY MANAGER [Marketing]

- Drive External Developers (API users)
- Social networking
- API Evangelist
- Administrate developer portal



TECH-LEAD [IT]

- Design & develop API resources
- Write API documentation
- Measure and improve API performance
- Write unit automated test





OPS [IT]

- Automated testing
- Automated deployment
- Scalability (elasticity) and SLA

API management.

API Management solutions generally offer the four following features:

- 1. API MANAGEMENT PORTAL**
Users enrolment
Publication / versioning
Usage statistics
Quotas
- 2. DEVELOPER PORTAL**
Self-enrolment
API Doc / Try-it interface
- 3. SECURITY**
API_KEY
OAuth2 / OIDC  complicated
- 4. API FAÇADE (ESB)**  use with caution

Common pitfalls

1. An API Management tool is not a Golden Hammer.

API Strategies are often summarised as “buying the right API Management product”. This reference card enumerates several aspects that should be addressed in an API strategy. These aspects concern three levels (functional, technical, organizational).

API Management products address at most 20% of API Strategies stakes.

2. Implementing an API façade with an API façade solution (ESB) as a target architecture.

Editors often focus on the façade feature (e.g. ESB, Gateway), but they should focus on Managing your API. Your API should be built with specific developments.

Build vs Buy.

You should distinguish between building your API from managing your API.

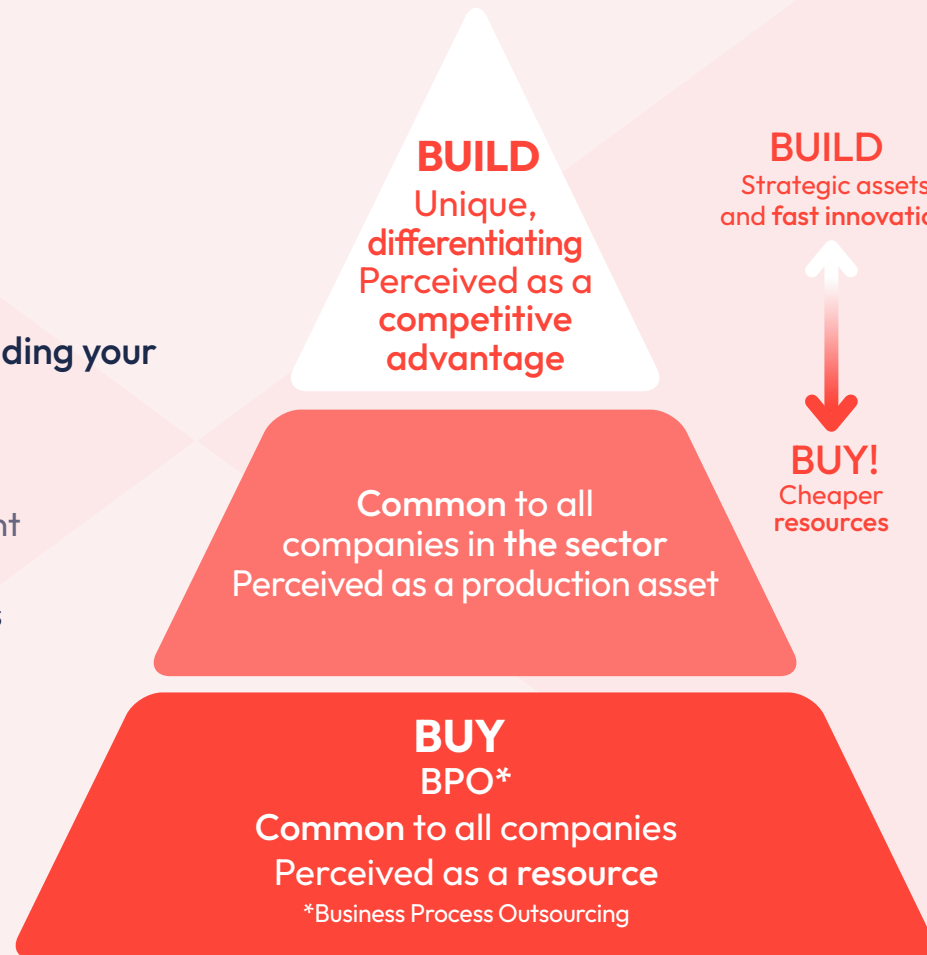
API - BUILD

The API becomes the main entry point to your CORE IT

- > Critical & differentiating components
- > A Key to a competitive advantage
- ESB are ineffective to build good API

PORTALS & SECURITY - BUY

- > API Management portal
- > Developer portal
- > Security



API integration patterns.

TO CREATE YOUR API, THERE ARE TWO INTEGRATION PATTERNS TO CONSIDER:

1. Façade pattern

The “façade” pattern is used to provide a simplified interface (API) to hide the complexity of your system. This pattern can actually be implemented in two ways: with a product (Buy) or with custom development (Build)

2. Microservices pattern (a key feature of Web-Oriented-Architecture)

PRE-BUILT FAÇADE PATTERN	CUSTOM FAÇADE PATTERN	MICROSERVICES PATTERN
Use an API Management (actually an ESB) to expose a REST API based on existing services.	Develop a custom App (Proxy) that will provide a REST API based on existing services.	This pattern consists to rebuild progressively your existing referentials to expose a pure RESTful API, directly from your CORE IT systems.
<ul style="list-style-type: none"> + Short time to market (good for a MVP) 	<ul style="list-style-type: none"> + Short time to market (good for a MVP). + Not dependent on an editor. + Will handle the complexity of your business logic. 	<ul style="list-style-type: none"> + Not dependent on an editor. + Will handle the complexity of your business logic. + No performance overhead.
<ul style="list-style-type: none"> - Dependent on the API Management/ ESB editor. - May not handle the complexity of your business logic. - A performance overhead should be considered. - The API Management/ESB and your existing service become highly coupled. 	<ul style="list-style-type: none"> - A performance overhead should be considered. - The façade and your existing services become highly coupled. 	<ul style="list-style-type: none"> - Not time to market for your API MVP.

WHICH PATTERN SHOULD I USE?

In most cases, a “façade pattern” is actually an anti-pattern, that was widely used to implement SOA with ESB.

You should consider façade pattern as a transitional solution, never as a final one. The façade constrains are:

- The resulting API will be limited by your underlying back-end services: “A great API on bad services is lipstick on a pig”!
- A bottleneck API team not adapted to “scale API”.
- The team will not handle the complexity of CORE IT business logic.
- The façade will be tightly coupled to existing back-ends.
Ex: if a CORE system evolves, the façade is impacted.
- With time, the API façade will become a huge monolith hard to maintain.
- Response time overhead.



OCTO Technology

▶ CABINET DE CONSEIL ET DE RÉALISATION IT ◀

“ Dans un monde complexe aux ressources finies, nous recherchons ensemble de meilleures façons d'agir. Nous œuvrons à concevoir et à réaliser les produits numériques essentiels au progrès de nos clients et à l'émergence d'écosystèmes vertueux”

- Manifeste OCTO Technology -

*There
is
a Better
Way*

1000 OCTOS 		Certified  Corporation®  autre cercle
 3 IMPLANTATIONS Paris Toulouse Hauts-de-France	OCTO EN TÊTE DU PALMARÈS 6x  GREAT PLACE TO WORK®	3 CONFÉRENCES  Unexpected Sources of Inspiration DUC-K CONF La conférence tech par OCTO  School of Product
FORMATION octo academy Learn to Change		



Conçu, réalisé et édité par OCTO Technology.

© OCTO Technology 2016

Les informations contenues dans ce document présentent le point de vue actuel d'OCTO Technology sur les sujets évoqués, à la date de publication. Tout extrait ou diffusion partielle est interdit sans l'autorisation préalable d'OCTO Technology.

Les noms de produits ou de sociétés cités dans ce document peuvent être les marques déposées par leurs propriétaires respectifs.

