

L'Architecture Applicative : le découplage des systèmes

Architecture d'Entreprise

L'Architecture Applicative : le découplage des systèmes

Architecture d'Entreprise

III — LE CARACTÈRE

« Conway's Law strikes back: organizations which design systems are constrained to produce designs which are copies of the communication structures of these organizations. » — Melvin Conway, *How Do Committees Invent?* (1968)

« Un catalogue d'APIs n'est pas une architecture applicative. »

Vidéo d'introduction

📄 Télécharger cette leçon en PDF

La tour de spaghetti qui a failli couler Industrias Yara

En 2017, Industrias Yara, ETI industrielle de 2 800 salariés, connaît une croissance rapide après l'acquisition de deux concurrents. La DSI, jusqu'alors organisée autour de trois applications principales (un ERP maison, un CRM maison, une GED maison), se retrouve à devoir intégrer six nouvelles applications métiers issues des acquisitions — chacune avec ses propres connecteurs, ses propres formats de données, ses propres conventions de nommage.

Au bout de dix-huit mois, l'architecture applicative ressemble à un plat de spaghetti. Chaque application parle à cinq autres applications, parfois via des flux de fichiers CSV déposés sur un partage réseau, parfois via des web services dont personne ne maintient la documentation, parfois via des triggers SQL dont seuls deux développeurs connaissent l'existence. Quand un développeur part à la retraite, deux projets d'intégration tombent en panne.

Camille, l'architecte applicatif recrutée en urgence, fait trois constats sans complaisance :

1. **Il n'y a pas d'architecture applicative, seulement un historique.** Les choix applicatifs ont été faits par sédimentation, en réaction à des urgences, sans vision d'ensemble. Le résultat est cohérent avec l'histoire de l'organisation, mais pas avec ses besoins présents.
2. **Personne ne sait ce qui dépend de quoi.** La cartographie applicative est obsolète depuis cinq ans. Personne n'a une vision complète des interfaces entre systèmes. Quand on demande « qui appelle l'API X ? », les réponses sont incomplètes ou contradictoires.
3. **Le pattern « Just a Bunch Of Web Services » (JBOWS) est partout.** Chaque équipe a créé ses propres APIs sans gouvernance transverse. Résultat : des APIs redondantes (cinq APIs pour la donnée client), des APIs sans contrat d'interface, des APIs orphelines (personne ne les utilise mais personne ne les décommissionne).

Camille passe six mois à reconstruire une **cartographie applicative** à jour, à établir des **contrats d'interface** pour les APIs critiques, et à mettre en place un **catalogue de services** qui distingue les APIs techniques des services métier. Le chantier n'élimine pas la complexité — il la rend visible et gérable.

C'est ce que cette leçon cherche à vous transmettre : **comment passer d'une tour de spaghetti à une architecture**

applicative gouvernée, sans naïveté sur ce que ce travail peut (et ne peut pas) accomplir.

La cartographie applicative d'Industrias Yara, organisée par niveaux de criticité, illustre comment une organisation peut passer d'une vue en spaghetti à une vue en strates gouvernables :

Premier principe — JBOWS n'est pas de l'architecture, c'est un anti-pattern

Le terme « Just a Bunch Of Web Services » désigne une collection d'APIs sans gouvernance transverse. Cette collection n'est pas une architecture — c'est un anti-pattern.

Sam Newman (*Building Microservices*, 2015) a popularisé l'expression JBOWS pour dénoncer la fausse sophistication de certaines architectures distribuées. Une collection de 200 APIs mal gouvernées n'est pas plus « moderne » qu'un monolithe bien structuré. Elle est souvent **moins** performante, **moins** fiable, et **plus** coûteuse à maintenir.

Les signes distinctifs d'un système JBOWS :

- Pas de contrat d'interface explicite — chaque API documente ce qu'elle veut
- Pas de propriétaire identifié pour chaque API — quand l'API casse, personne ne réagit
- Pas de versionning cohérent — les breaking changes arrivent sans prévenir
- Pas de mesure d'usage — on ne sait pas si elle est utilisée par 1 ou 100 clients
- Pas de politique de décommissionnement — les APIs mortes restent actives indéfiniment

La contrefaçon classique consiste à croire que « passer aux microservices » résoudra ces problèmes. En réalité, **les microservices sans gouvernance ne font que multiplier le problème par N**. La parade est de mettre en place la gouvernance **avant** la multiplication des services.

Pour Industrias Yara, Camille a démarré par un **inventaire honnête** : 187 APIs identifiées, dont 67 sans propriétaire, 23 sans documentation, 8 sans aucun appelant connu sur les 90 derniers jours. Première action : décommissionner les 8 APIs orphelines. Résultat : moins de surface à maintenir, démonstration tangible que la gouvernance fonctionne.

***Et toi ?** Combien d'APIs ton organisation expose-t-elle ? Combien ont un propriétaire identifié ? Combien sont documentées ? Combien sont utilisées par au moins un client en production ?*

Deuxième principe — La cartographie applicative est vivante, pas commémorative

Une cartographie applicative qui n'est pas mise à jour régulièrement devient un mensonge documentaire — pire qu'une absence de cartographie, car elle induit les décideurs en erreur.

L'Open Group recommande que la cartographie applicative soit **maintenue activement**, avec :

- **Un propriétaire identifié** — un architecte applicatif (souvent à temps plein) qui maintient la cartographie comme une activité continue, pas comme un livrable de projet
- **Une source de vérité unique** — un outil partagé (ArchiMate, LeanIX, Mega HopEx) où la cartographie est éditée
- **Un cycle de revue** — trimestriel pour les applications critiques, annuel pour les autres
- **Des indicateurs d'usage** — chaque application est documentée avec son taux d'utilisation, son coût de run, son risque de fin de support éditeur

La contrefaçon classique consiste à produire une cartographie de 87 applications dans un PowerPoint de 200 pages, jamais mise à jour, oubliée sur un sharepoint. Cette cartographie-obélisque n'a aucune valeur opérationnelle. Elle a même une valeur **négative** : les décideurs croient qu'elle est à jour et prennent des décisions sur la base d'informations fausses.

Pour Industrias Yara, Camille a mis en place un **outil léger** : un tableur partagé entre cinq architectes applicatifs, avec une cellule « dernière mise à jour » par ligne. Le cycle de revue est mensuel pour les 20 applications critiques, trimestriel pour les autres. La cartographie a 90 % des informations vérifiées dans les 90 jours, contre 30 % avant la réforme.

***Et toi ?** À quand remonte la dernière mise à jour de ta cartographie applicative ? Quel pourcentage des applications sont documentées à jour ? Quel outil utilises-tu ?*

Troisième principe — Le contrat d'interface est un acte juridique, pas technique

Une API sans contrat d'interface est une API sans propriétaire. Le contrat n'est pas un fichier WSDL ou OpenAPI — c'est un engagement entre l'équipe qui produit l'API et celles qui la consomment.

Eric Evans (*Domain-Driven Design*, 2003) a popularisé la notion de **Bounded Context** : chaque service métier a une frontière claire, avec un langage et un contrat explicites. Cette notion est au cœur d'une architecture applicative mature.

Un contrat d'interface document devrait contenir au minimum :

- **Le périmètre fonctionnel** — quels cas d'usage l'API couvre, et (surtout) lesquels elle **ne couvre pas**
- **Le propriétaire** — un nom, une équipe, une adresse de support
- **La SLA** — temps de réponse attendu, taux de disponibilité, temps de résolution d'incident
- **Le contrat de données** — le schéma des entrées et sorties, documenté (OpenAPI, Protobuf, Avro)
- **La politique de versionning** — comment les breaking changes sont annoncés, combien de versions coexistent
- **La politique de dépréciation** — combien de temps une ancienne version reste supportée après une nouvelle

La contrefaçon classique consiste à publier une spécification OpenAPI et à croire que le contrat est signé. Sans les engagements de SLA, de versionning, et de dépréciation, le contrat reste incomplet — et l'équipe consommatrice ne sait pas ce qu'elle peut attendre.

Pour Industrias Yara, Camille a imposé un **template de contrat** de deux pages, signé conjointement par l'équipe qui produit l'API et un représentant des consommateurs. La signature n'est pas un acte bureaucratique — c'est un engagement de support.

***Et toi ?** Combien d'APIs dans ton organisation ont un contrat complet (périmètre, SLA, versionning, dépréciation) ? Combien n'ont que la spécification technique ?*

Quatrième principe — L'ERP est un système complexe, pas une boîte noire

L'ERP est le système qui concentre le plus de complexité — et le plus de risques. Le traiter comme une boîte noire qu'on « configure » est une illusion. Le comprendre comme un système complexe qu'on doit maîtriser est une nécessité.

L'ERP (SAP, Oracle, Microsoft Dynamics, etc.) cumule quatre caractéristiques qui en font un objet d'architecture à part entière :

1. **Il est critique** — la majorité des processus métiers passent par lui. Un incident ERP peut paralyser l'organisation.
2. **Il est paramétrique, pas configurable** — chaque paramétrage a des implications en cascade. « Configurer » un ERP sans comprendre son modèle de données produit des comportements imprévus.
3. **Il évolue avec l'éditeur** — les montées de version imposent des refontes de paramétrage. L'ERP n'est jamais stable.
4. **Il porte la dette technique** — les paramétrages successifs accumulent de la dette. Un ERP « vieux » de 15 ans peut être plus complexe à maîtriser qu'un ERP récent mal paramétré.

La contrefaçon classique consiste à externaliser l'ERP à un intégrateur et à perdre la maîtrise interne du système. L'organisation devient alors dépendante de l'intégrateur pour chaque évolution, et la dette technique s'accumule sans visibilité.

La parade est de maintenir une **équipe interne ERP** (même petite), avec :

- Un **architecte ERP** qui comprend le modèle de données
- Un **responsable de la maîtrise d'ouvrage** qui porte les besoins métiers
- Un **processus de revue** des paramétrages critiques (workflow, comptabilité, paie)

Pour Industrias Yara, Camille a créé un poste d'architecte ERP dédié — une décision qui a coûté 90 k€ par an et a évité des millions d'euros d'incidents et de projets mal cadrés.

Et toi ? Combien de personnes dans ton organisation maîtrisent le modèle de données de l'ERP ? Combien de paramétrages critiques ont été documentés ? Combien de fois l'intégrateur a-t-il été appelé pour un incident causé par un paramétrage obscur ?

Cinquième principe — Un catalogue de services n'est pas une liste d'APIs

Le catalogue de services est un outil de gouvernance métier, pas un annuaire technique. Il sert à répondre à la question « quel service métier est exposé par quelle application, sous quelle responsabilité ? » — pas « voici nos 200 URLs ».

Un catalogue de services mature contient, pour chaque service :

- **Le nom métier** — un nom compréhensible par un opérationnel (« service de souscription client »)
- **Le périmètre fonctionnel** — ce que le service fait, en langage métier
- **L'application qui l'expose** — nom de l'application, éditeur, version
- **L'API technique** — URL, contrat, version
- **Le propriétaire métier** — la direction responsable du service
- **Le propriétaire technique** — l'équipe qui maintient l'API
- **Les SLA** — engagements de performance et de disponibilité
- **Les consommateurs** — qui appelle ce service, dans quel but

La contrefaçon classique consiste à publier une **liste d'APIs** (URL + description technique) en la qualifiant de catalogue. Cette liste ne sert qu'aux développeurs, pas aux opérationnels ni aux décideurs. Le catalogue de services sert à **rendre l'architecture applicative discutable au niveau métier**.

Pour Industrias Yara, Camille a mis en place un catalogue à deux niveaux : un **catalogue métier** (15 services, lisible par un COMEX) et un **catalogue technique** (187 APIs, réservé aux architectes). Les deux catalogues sont liés, mais leur

public est différent.

***Et toi ?** Combien de services métier sont exposés par ton SI ? Combien ont un propriétaire métier identifié ? Combien ont une SLA publiée ?*

Application — Ma cartographie applicative

À produire séance tenante. Cinq sections précises.

1. **Inventaire** — Liste exhaustive des applications de ton organisation (nom, éditeur, version, date de mise en production).
2. **Critique** — Pour chaque application, note de criticité (1 à 5), avec la justification.
3. **Contrats** — Pour les 10 applications les plus critiques, identification des 5 principaux contrats d'interface (fournisseur ou consommateur).
4. **APIs orphelines** — Liste des APIs sans propriétaire, sans usage, ou sans documentation. Cible de décommissionnement à 90 jours.
5. **Catalogue métier** — Liste des 10 à 15 services métier exposés, avec leur propriétaire métier nommé.

|> Cette cartographie est ton premier livrable d'architecture applicative. Elle est la base de toute gouvernance ultérieure.

Module complémentaire — L'API-first comme principe architectural

Dans un SI moderne, l'application n'est plus un silo. Elle est un **point d'entrée** dans un écosystème de services. La question n'est plus « quelle application possède cette donnée ? » mais « **quelle API expose cette donnée, et à quelles conditions ?** ».

Le principe API-first

L'**API-first** est une approche où les **interfaces** (APIs) sont conçues *avant* les implémentations. Ce n'est pas un choix technique : c'est un **principe architectural** qui transforme la gouvernance applicative.

Trois implications concrètes :

1. **L'API est un produit, pas un canal.** Elle a un propriétaire nommé, un cycle de vie, une documentation, une SLA de latence et de disponibilité, un versioning, une politique de dépréciation.
2. **Le contrat d'interface précède l'implémentation.** Les consommateurs peuvent s'intégrer avant que l'implémentation ne soit prête (par mock, par stub), ce qui parallélise le développement et clarifie les attentes.
3. **L'API est l'unique vecteur d'accès aux données.** Aucun accès direct aux bases de données (SQL direct) n'est toléré. Cette règle, héritée du cas Retail (L4 — Schéma Directeur Omnicanal), est un garde-fou de **sécurité** et de **cohérence**.

Le « Single Source of Truth » et la gouvernance des données

L'API-first rejoint le principe du **Single Source of Truth** (SSOT) : pour chaque donnée critique, il existe **une seule API de référence** qui fait foi. Toutes les autres sources (réplicas, caches, bases de shadow IT) doivent se synchroniser

depuis cette API. Sans ce principe, l'organisation cumule les versions divergentes de la même information — et les erreurs qui en découlent.

Chez Mutuelle Plus, l'absence de SSOT sur la donnée client (Gest+ + 50 tableurs Excel en agence) génère les doublons qui plombent le délai de remboursement. La mise en place d'une API « Client » comme unique source de vérité est une **condition préalable** à la refonte du SI Décisionnel (voir L7 — Architecture des Données, cas pratique Mutuelle Plus).

Trois erreurs de gouvernance API-first à éviter

1. « **API-flood** » : exposer une API pour chaque besoin sans regroupement logique. Le consommateur se retrouve à intégrer 50 APIs pour un seul processus métier. La règle : une **API métier** doit couvrir un **domaine fonctionnel** (ex. : API « Contrats »), pas une seule table.
2. **API sans propriétaire** : l'API est publiée mais personne n'est nommé responsable. Quand un incident survient, personne ne répond. Solution : un **registre des APIs** avec, pour chacune, un Product Owner métier, un Tech Lead, et un SLA documenté.
3. **Pas de politique de dépréciation** : les APIs s'accumulent, certaines ne sont plus consommées mais continuent à coûter (maintenance, hébergement, monitoring). Planifier une revue trimestrielle des APIs sous-utilisées et décommissionner.

L'outillage minimum

Une gouvernance API-first efficace repose sur quatre composants :

- **Un catalogue d'APIs** (Backstage, Stoplight, ReadMe) qui sert de portail unique aux développeurs.
- **Une API Gateway** qui centralise l'authentification, le throttling, le monitoring, et la journalisation.
- **Des standards de description** (OpenAPI / Swagger pour REST, AsyncAPI pour les flux événementiels).
- **Un registre de schémas** qui garantit la compatibilité ascendante et gère les évolutions de contrats.

Synthèse

L'API-first n'est pas un projet technique. C'est un **changement de paradigme** : l'organisation passe d'une logique de silos applicatifs à une logique de **services réutilisables**. Cela nécessite un investissement initial (catalogue, gateway, formation) et une discipline continue (revues, dépréciations). Le ROI se mesure en **vitesse d'intégration** (nouveau partenaire en moins de quatre semaines, comme dans le cas Retail) et en **réduction des incohérences** (zéro doublon dans la donnée client, comme dans le cas Mutuelle Plus).

Encadré Au lecteur / À l'animateur

Question d'ouverture : « *Combien d'APIs ton organisation expose-t-elle aujourd'hui ? Combien ont un propriétaire identifié ?* »

Profils à observer :

- **Le microservice-maniaque**. Croit que la multiplication des APIs résoudra les problèmes. *Question* : « *Combien d'APIs as-tu sans propriétaire ?* »
- **Le collectionneur de cartographies**. Produit une cartographie PowerPoint de 200 pages, jamais mise à jour. *Question* : « *À quand remonte la dernière mise à jour ?* »
- **Le dépendant de l'intégrateur**. A externalisé l'ERP et perdu la maîtrise interne. *Question* : « *Combien de personnes maîtrisent le modèle de données ?* »
- **L'API-phobe**. Refuse de documenter ses APIs par peur de l'engagement. *Question* : « *Combien de consommateurs ont signé un contrat avec toi ?* »

Pièges d'animation :

- Ne pas laisser la discussion dériver vers les outils (Kubernetes, Istio, etc.). Rappeler que l'architecture applicative est d'abord une question de gouvernance.
- Ne pas confondre API et service. Une API est une implémentation technique ; un service est un engagement métier.

Ce que la leçon 9 vous demande

Ouvrez votre dossier d'architecte à la page suivante. Préparez :

- **Une cartographie applicative** à jour, avec criticité notée.
- **10 contrats d'interface** documentés pour les applications critiques.
- **Un catalogue métier** de 10 à 15 services, avec propriétaires nommés.

Sources de cette leçon : TOGAF Series Guide to Microservices Architecture (G243) · ArchiMate 3.2 Specification · Eric Evans, *Domain-Driven Design* (2003) · Sam Newman, *Building Microservices* (2015) · Cas Industrias Yara (anonymisé)

Leçon suivante : **L9 — L'Architecture Technique : les services de plate-forme.**

Quiz — 18 questions pour valider votre compréhension

Trois séries de six questions. Prenez un crayon, écrivez vos réponses dans l'espace prévu. Score indicatif : 14/18 minimum pour valider la leçon.

A — COMPREHENSION

A1 Qu'est-ce que l'anti-pattern JBOWS ? Pourquoi la multiplication des APIs sans gouvernance est-elle contre-productive ?

A2 Pourquoi une cartographie applicative doit-elle être vivante (maintenue activement) plutôt qu'un livrable de projet commémoratif ?

A3 Qu'est-ce qu'un contrat d'interface complet (périmètre, SLA, versionning, dépréciation) ? Pourquoi la spécification technique OpenAPI seule ne suffit-elle pas ?

A4 Quelles sont les quatre caractéristiques qui font de l'ERP un objet d'architecture à part entière ? Pourquoi faut-il maintenir une équipe interne ?

A5 Quelle est la différence entre un catalogue de services et une liste d'APIs ? À quel public chacun s'adresse-t-il ?

A6 Qu'est-ce que la loi de Conway ? En quoi est-elle pertinente pour l'architecture applicative ?

B — AUTO-EVALUATION

B1 Combien d'APIs ton organisation expose-t-elle ? Combien ont un propriétaire identifié ? Combien sont documentées ? Combien sont utilisées par au moins un client en production ?

B2 À quand remonte la dernière mise à jour de ta cartographie applicative ? Quel pourcentage des applications sont documentées à jour ? Quel outil utilises-tu ?

B3 Combien d'APIs dans ton organisation ont un contrat complet (périmètre, SLA, versionning, dépréciation) ? Combien n'ont que la spécification technique ?

B4 Combien de personnes dans ton organisation maîtrisent le modèle de données de l'ERP ? Combien de paramètres critiques ont été documentés ?

B5 Combien de services métier sont exposés par ton SI ? Combien ont un propriétaire métier identifié ? Combien ont une SLA publiée ?

B6 Sur ton dernier projet d'architecture, as-tu rencontré une tour de spaghetti ? Combien d'APIs as-tu identifiées sans propriétaire ?

C — ENGAGEMENTS A 30/45/60 JOURS

C1 Produire un inventaire exhaustif des applications de mon organisation (nom, éditeur, version, date de mise en production). (Échéance : J+14.)

C2 Attribuer une note de criticité (1 à 5) à chaque application avec justification. (Échéance : J+21.)

C3 Identifier les 10 applications les plus critiques et documenter leurs 5 principaux contrats d'interface (fournisseur ou consommateur). (Échéance : J+45.)

C4 Lister les APIs orphelines (sans propriétaire, sans usage, ou sans documentation) et planifier leur décommissionnement. (Échéance : J+30.)

C5 Construire un catalogue métier de 10 à 15 services, avec leur propriétaire métier nommé. (Échéance : J+60.)

C6 Mettre en place un cycle de revue trimestriel pour les 20 applications les plus critiques. (Échéance : J+90.)
