R5.A.08 :
Qualité de développement

Chouki Tibermacine
Chouki.Tibermacine@univ-ubs.fr

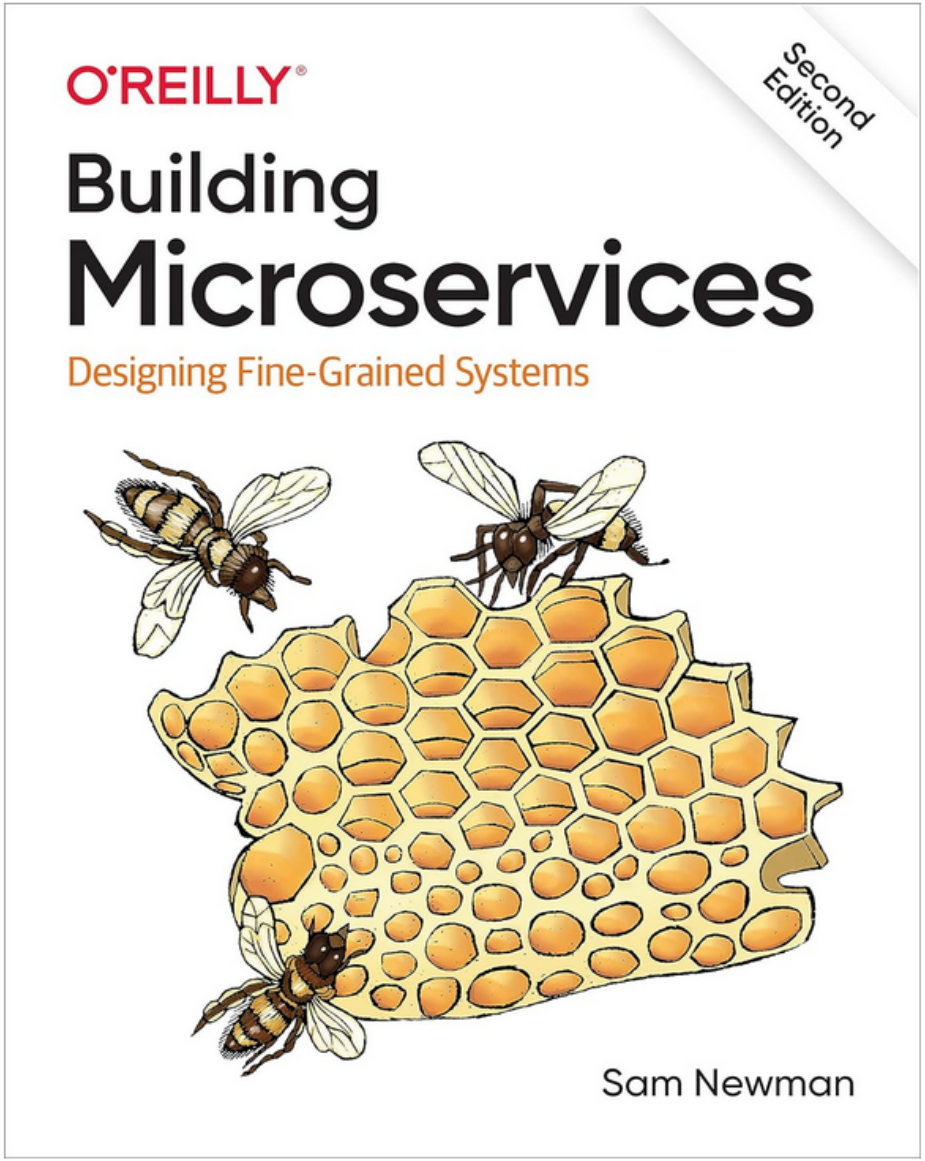
Plan prévisionnel de la ressource

1. Intro aux architectures logicielles
2. Documentation d'architectures en UML
3. Styles et patrons d'architectures
4. Architectures à microservices
5. Design & Implem de frameworks
6. Patrons de conception : Façade, Bridge, MVC et MVVM
7. Patrons de conception : Builder, Proxy et Visitor
8. Autres patrons

Plan prévisionnel de la ressource

1. Intro aux architectures logicielles
2. Documentation d'architectures en UML
3. Styles et patrons d'architectures
4. Architectures à microservices
5. Design & Implem de frameworks
6. Patrons de conception : Façade, Bridge, MVC et MVVM
7. Patrons de conception : Builder, Proxy et Visitor
8. Autres patrons

Référence bibliographique

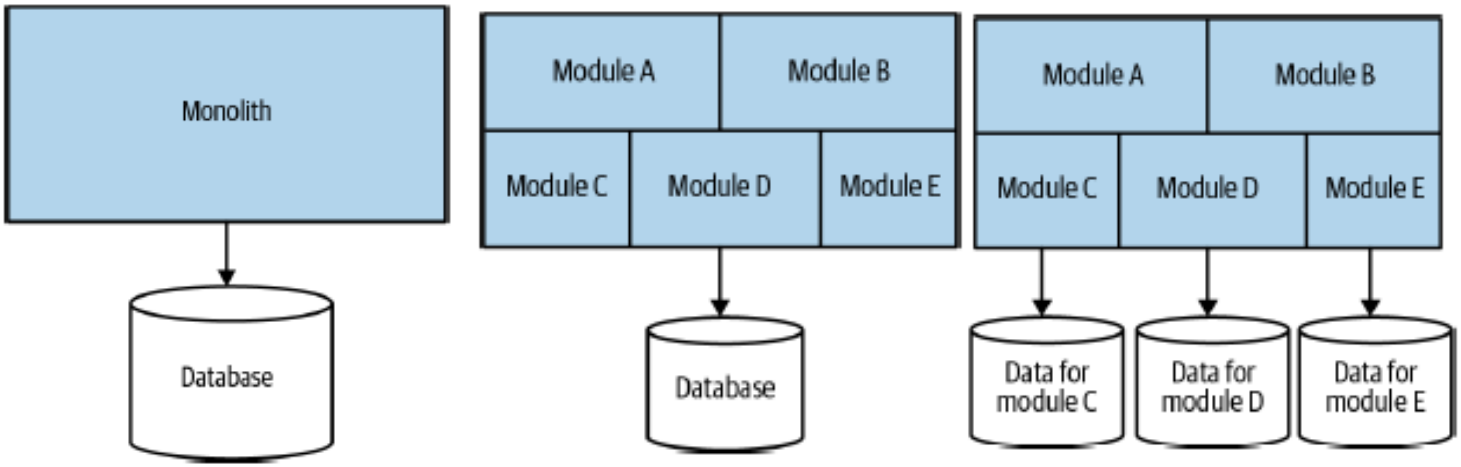


Plan du cours

1. Évolution des styles d'architecture Web
2. Architectures à microservices (MSA)
3. Design de MSA
4. Communication inter-microservices

Applications monolithiques

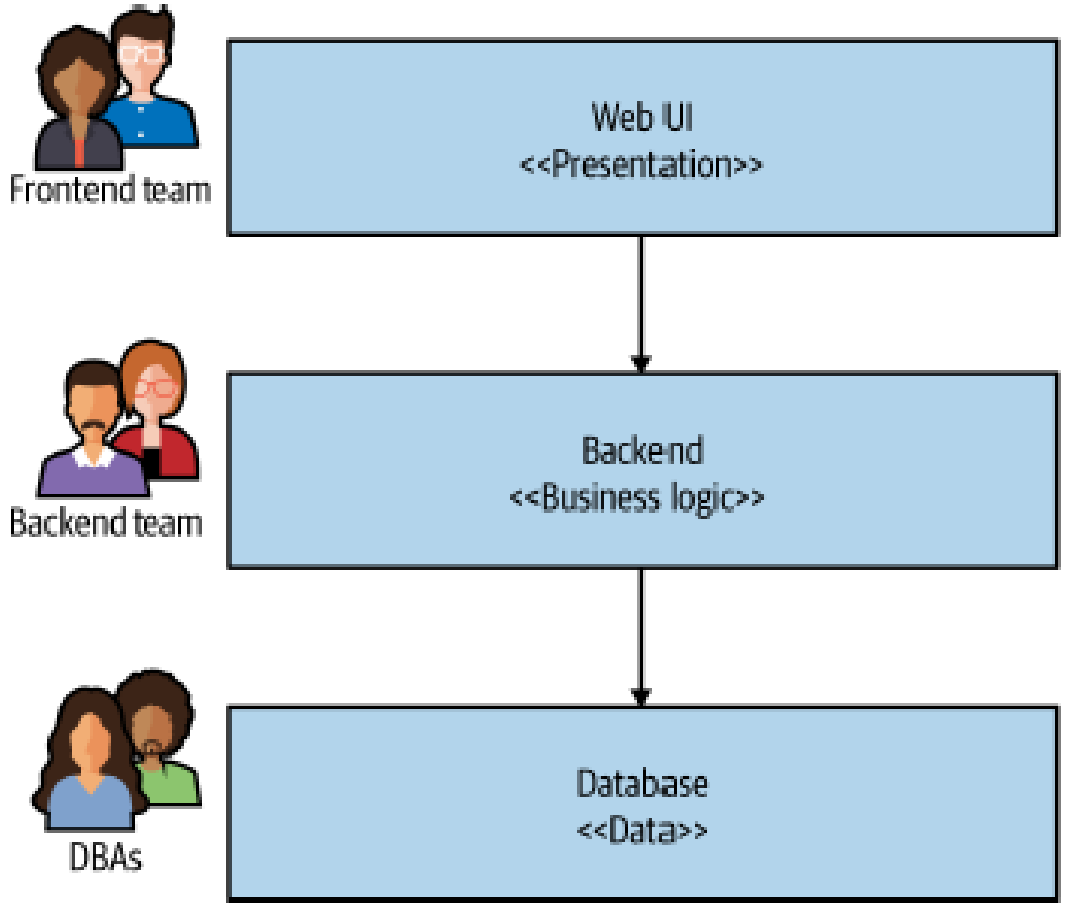
- Applications déployées en un seul gros bloc (d'où le nom)
- Différentes sortes :
  - monolithe à processus unique (fig. ci-dessous à gauche)
  - monolithe modulaire (les 2 autres figures ci-dessous)
  - monolithe distribué : comprend +ieurs services, mais déployés ensemble



Applications monolithiques -suite-

- Défauts majeurs :
  - les différentes parties de l'app sont dépendantes les unes des autres au build, à la livraison, au déploiement, ...
  - *Delivery Contention* : conflits d'ownerships sur des bouts de code proches
- Elles ont toutefois les avantages de :
  - simplicité de déploiement
  - simplicité de réutilisation de code (simples bibliothèques)
- Avoir un monolithe ne veut pas toujours dire : disposer d'un code legacy à moderniser
- ça peut être le style d'architecture par défaut. Faire du microservice doit être justifié

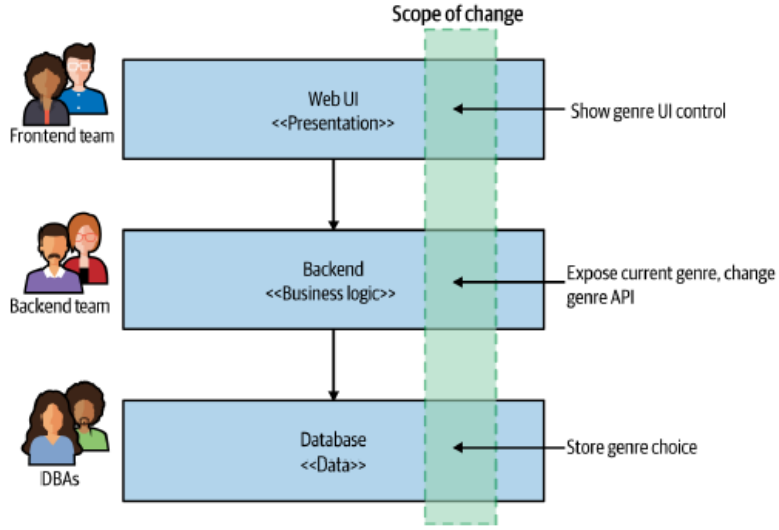
Architectures multi-couches (*Layered Architectures*)



Architectures multi-couches (*Layered Architectures*)

Défauts :

- Un changement porte souvent sur l'ensemble des couches
- Cela implique parfois plusieurs équipes



Architectures à services (SOA)

- Style ancêtre des architectures à microservices
- Style plutôt théorique préconisant la conception d'applications en termes de *service providers* et *service consumers* découplés, dépendant les uns des autres à travers des interfaces de services (API = contrats)
- Parfois on s'appuie sur un annuaire de services pour rechercher un *service provider*
- Ces services sont souvent distribués (accessibles sur le réseau via des protocoles, comme SOAP, Websockets ou HTTP)

Architectures à services (SOA)

Défauts :

- Services déployés ensemble, avec souvent des bases de données partagées
- Dans une même application, parfois, juste une partie est implémentée comme SOA. Le reste étant monolithique (souvent modulaire)
- Granularité des services ambigüe, souvent très fine, concernant juste une fonctionnalité partagée, ayant une API et plusieurs implémentations possibles
- Pas de recommandations claires sur la définition des frontières des services

Plan du cours

1. Évolution des styles d'architecture Web
2. Architectures à microservices (MSA)
3. Design de MSA
4. Communication inter-microservices

Qu'est-ce qu'un micro-service ?

Un module logiciel :

1. Déployable indépendamment des autres : quand on le déploie, on n'a pas besoin de déployer autre chose à côté
2. Modélisé autour d'un domaine métier
3. Possède son propre état (gère sa propre base de données)

- De quelle taille ? Cela importe peu (le mot "micro" à ignorer)
  Complexité d'un MS doit être gérable
- ça apporte de la flexibilité ? "*microservices buy you options*"
  (James Lewis, Thoughtworks, GB)
  L'adoption des MS doit se faire de façon progressive

Avantages des microservices

- Hétérogénéité des technologies : chaque microservice peut être développé avec un langage de programmation différent, peut avoir sa base de données du type le plus approprié pour les données (graphe, document, relationnel, ...)
- Robustesse : 1 MS qui échoue peut être isolé, l'app peut continuer à fonctionner normalement
- Passage à l'échelle (*Scaling*) : Possibilité de répliquer les services critiques ⇒ coût moindre que de répliquer un monolithe

Avantages des microservices -suite-

- Déploiement facilité : en cas de changement, déploiement par microservice ⇒ moins de risque (rollback facile aussi)
- Organisation plus efficace : de plus petites équipes travaillant sur de petites bases de code (microservices) sont plus productives ⇒ alignement Architecture-Organisation
- Réutilisabilité et composabilité : possiblité d'assembler les microservices de différentes façons pour customizer des applications (microservices ≈ pièces de Lego)

Points difficiles à gérer avec les microservices

- Expérience des développeurs avec les MS : sur un environnement de Dev, si on a beaucoup de MS à démarrer pour une application (chacun utilisant une JVM, par ex), ça peut vite devenir compliqué
- Surcharge de technologies : beaucoup d'options possibles (ne pas se faire submerger et rester sobre, du moins au début)
- Coût initial lors de l'adoption : habitudes des équipes à changer, plus de ressources à utiliser
- Gestion des logs : à agréger avec des outils (lire : `https://geekflare.com/open-source-centralized-logging/`)
- Gestion du reporting sur différentes sources de données (solutions : *real-time streaming*, publication des données dans une BdD centrale de reporting, ...)

Points difficiles à gérer avec les microservices -suite-

- Gestion plus fine du monitoring et de l'observabilité
- Sécurité à gérer plus finement (MSA = application distribuée avec données sur le réseau)
- Tests de petite portée (unitaires, par ex) facilité, mais tests end-to-end difficiles à réaliser (tout doit être déployé, sources de défaillances multiples lors des tests)
- Latence : remplacement des appels de fonctions par de l'IPC (*Inter-Process Communication*), le passage d'arguments par sérialisation/déserialisation d'objets, ...
  ⇒ mais ça peut être toléré dans beaucoup de cas
- Cohérence des données : plusieurs bases de données à gérer pour une même application (plus de transactions possibles)
  ⇒ utiliser des solutions comme les sagas

Plan du cours

1. Évolution des styles d'architecture Web
2. Architectures à microservices (MSA)
3. Design de MSA
4. Communication inter-microservices

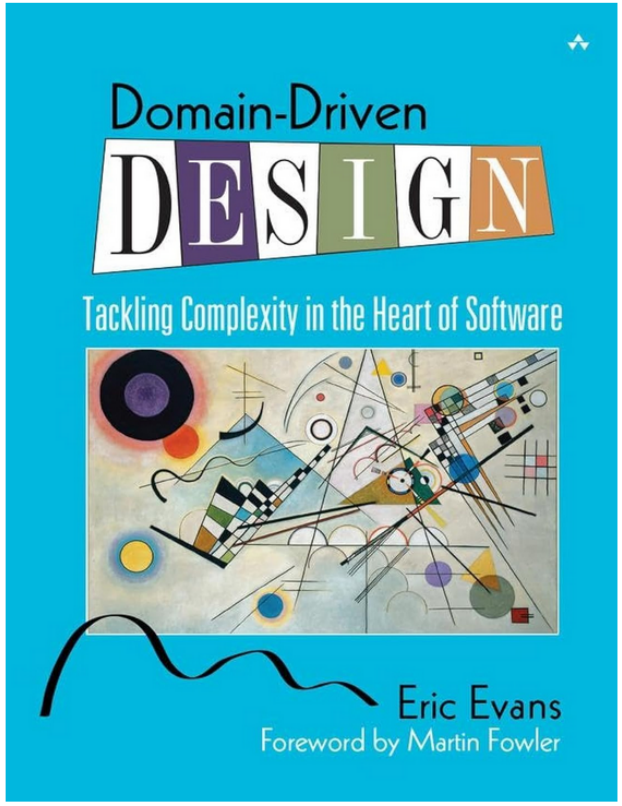
Frontières des microservices

- Nous devons être capable de changer, déployer et releaser la fonctionnalité d'un MS de façon indépendante
- Les solutions existent dans la décomposition modulaire :
  - Encapsulation des données (*Information Hiding*) : ne partager comme données que ce qui doit l'être ⇒ temps de dév amélioré (parallélisation), compréhensibilité et flexibilité
  - Cohésion forte (intra-MS) : le code qui change ensemble doit rester ensemble
  - Couplage faible (inter-MS) : un changement dans le code d'un MS ne doit pas nécessiter un changement dans un autre
    ⇒ *No Chatty Communication*

*"A structure is **stable** if cohesion is strong and coupling is low"*
(*Larry L. Constantine*)

Domain-Driven Design (DDD)

- Méthodologie de conception de logiciels, dirigée par le domaine métier
- Introduite dans le livre d'Eric Evans
- Identifier les frontières des MS par analyse du domaine métier
  $\Rightarrow$ Matching : code $\rightleftarrows$ monde réel
- Concepts clés :
  1. *Ubiquitous Language*
  2. *Aggregate*
  3. *Bounded Context*



Ubiquitous Language

- Se forcer à utiliser le même vocabulaire dans le code que celui pratiqué par les utilisateurs
- Dans tout artefact, pratiquer le même vocabulaire (*user stories*, doc technique, code, ...)
- Utiliser le même vocabulaire (qui doit être minimaliste) simplifie la modélisation du monde réel
- Cela simplifie aussi la communication entre les différents protagonistes : *business analysts* ou les *product owners*, les dev, ...

Aggregate

- Théoriquement, un aggregate est une collection d'objets représentant de petites unités issues de la base de données (définition un peu ambiguë)
- En pratique, c'est une représentation d'un concept du domaine métier : une commande, une facture, un produit, ...
- Généralement, un aggregate a un cycle de vie ⇒ implémentable donc comme une machine à états
- Exemple : un item dans une commande n'a de sens que lorsqu'il fait partie d'une commande. Donc, une commande est un aggregate (pas l'item dans une commande)
- Le code qui gère les transitions des états d'un aggregate et l'état lui-même doit être centralisé

Aggregate -suite-

- Un aggregate doit donc être géré par un seul MS
- Un MS peut toutefois gérer plusieurs aggregates
- L'accès à ces aggregates se fera via l'interface du MS (encapsulation)
- Des aggregates peuvent avoir des relation entre eux (Client lié à une Commande et une ListeDeVoeux). Ces relations doivent être gérées par les MS qui les encapsulent ou d'autres MS
- Relations entre aggregates intra-MS peuvent être gérées par des clés étrangères de BdD
- Relations entre aggregates inter-MS peuvent être gérées en stockant des ID ou des URI d'API REST

Bounded Context

- Un bounded context est un ensemble d'aggregates avec une responsabilité (interface en termes de fonctionnalités) explicite
- Un bounded context cache les détails d'implémentation
- **Un microservice = un aggregate ou un bounded context**
- Découpage possible : considérer les bounded contexts comme MS, puis décomposer en MS plus fins (aggregates, pas plus fin que ça) si trop grands
- Un processus possible pour mener cela : *Event Storming* `https://www.eventstorming.com/`
- D'autres critères de décomposition exceptionnels : les données (leur sécurité et privacy), le choix des technologies, organisationnels (équipes "propriétaires" de MS)

Migrer un monolithe vers les MS

- La migration doit être justifiée (scalabilité, par ex)
- La migration doit être menée de façon incrémentale
  $\Rightarrow$ le big bang ne marche pas
- Commencer par des petits morceaux (un goulot d'étranglement dans l'app, par ex) faciles
- Utiliser des outils d'analyse de code (CodeScence par ex) pour identifier les parties volatiles dans le code (qui évoluent beaucoup), les parties avec de trop fortes dépendances, ...
- Ensuite, décomposition par couche (UI, backend & data)

Patterns de migrations

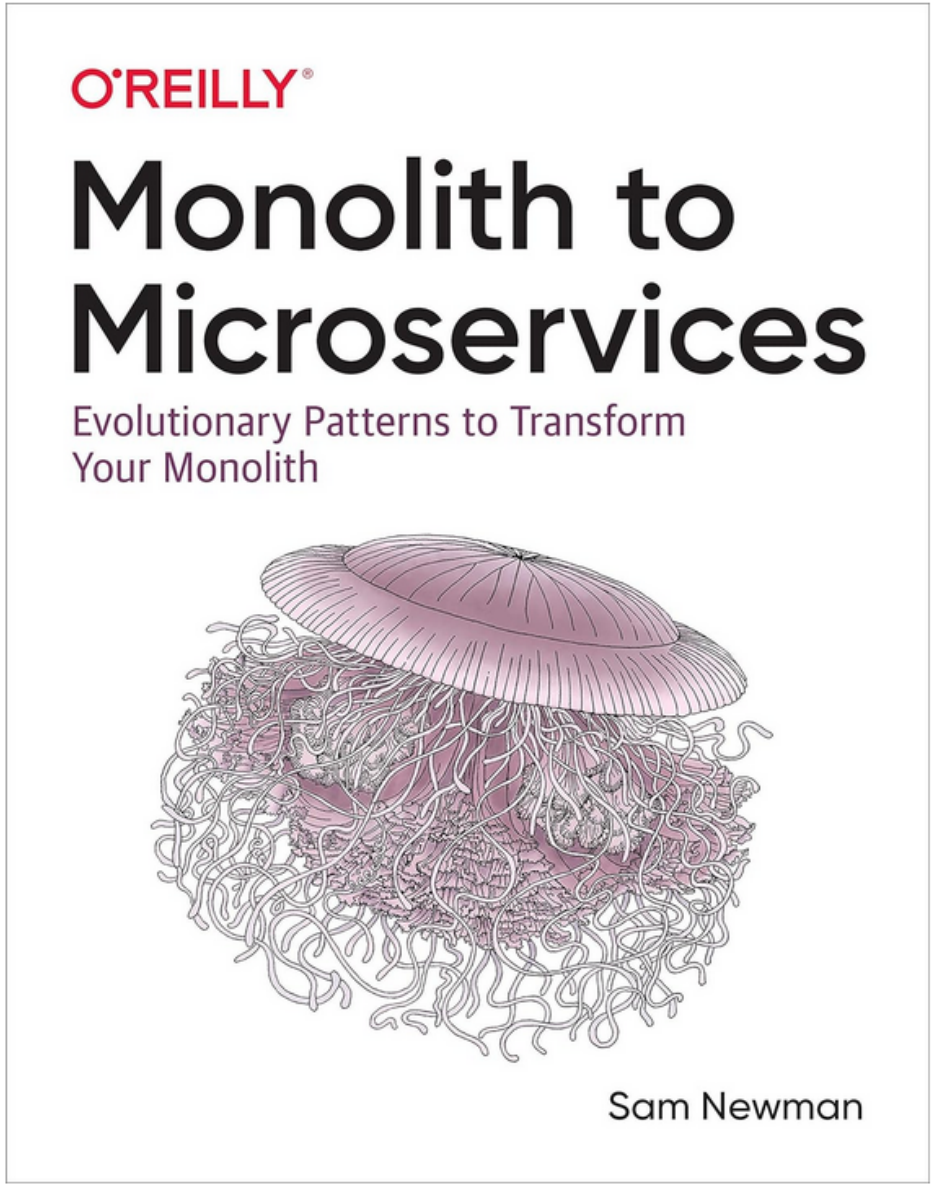
- *Strangler Fig Pattern* : envelopper l'app monolitique existante petit à petit (par des MS) jusqu'à la remplacer complètement
- *Parallel Run* : les deux variantes s'exécutent en parallèle
- *Feature Toggle*, grâce aux Feature Flags :

  `https://www.getunleash.io/`

Points de vigilance

- **Performances de la base de données :** par ex, requêtes SQL avec jointures (clés étrangère dans une même BdD) ⇒ requête SQL, suivie d'une requête HTTP, puis enfin une requête SQL (ID ou URI qui fait référence à une autre table dans un autre MS)
- **Intégrité des données :** par ex, suppression de données doivent être propagées par les MS (ce n'est plus du ressort du serveur de BdD)
- **Transactions :** transactions distribuées ou *sagas*
- **Reporting de données sur des BdD distribuées :** créer une BdD dédiée au reporting, alimentée/synchronisée par les MS

Pour plus de détails



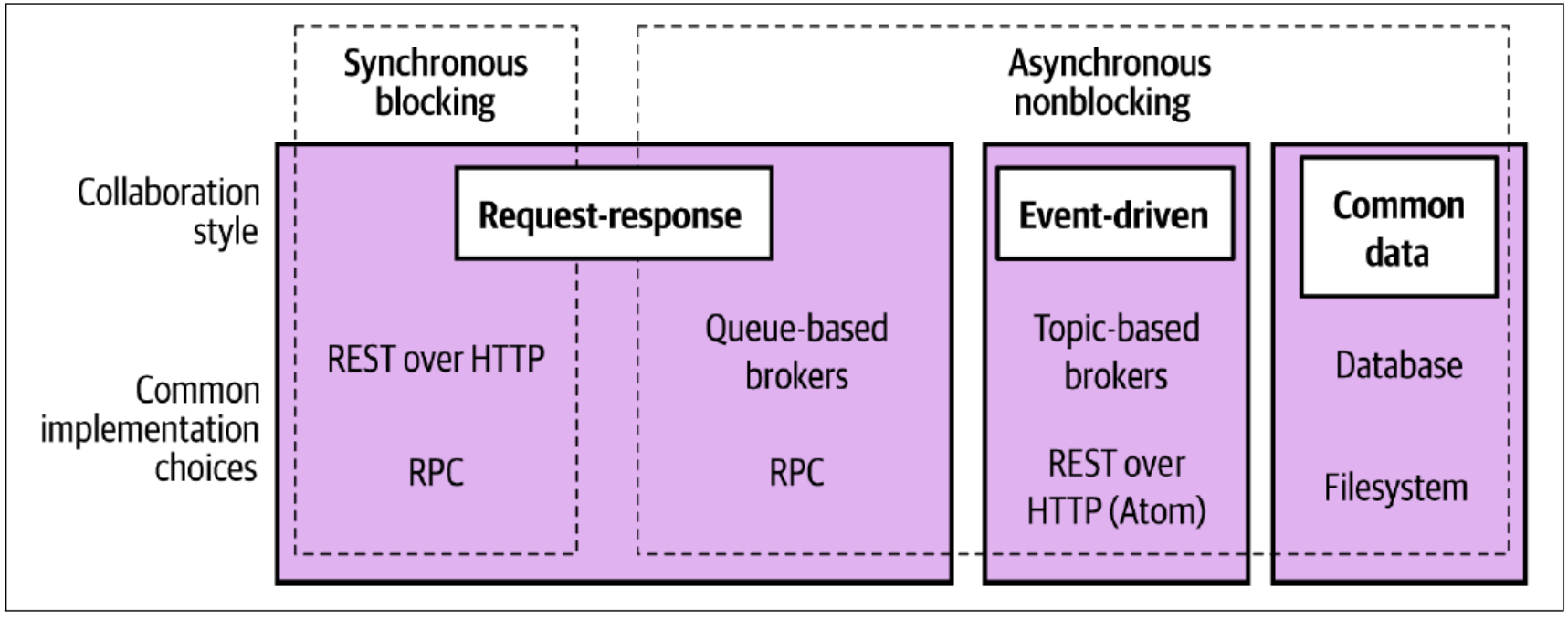
Plan du cours

1. Évolution des styles d'architecture Web
2. Architectures à microservices (MSA)
3. Design de MSA
4. Communication inter-microservices

D’une communication intra-processus à une IPC

- Appels de fonctions (optimisés par les compilateurs/runtime) dans le même espace mémoire vers des sockets, files de messages, ...
- Performances moins bonnes d’où la nécessité de revoir parfois les APIs : grouper des fonctions dans 1 seule, revoir les paramètres de type “grosses structures de données” (du passage par référence/adresse à de la sérialisation, ...), ...
- Changement des interfaces plus délicats : simple refactoring dans un monolithe pour répercuter un changement vs évolution d’une API REST
- Gestion des erreurs : codes d’erreurs ou exceptions dans le monolithe vs codes d’erreurs HTTP (400 et 500) à propager

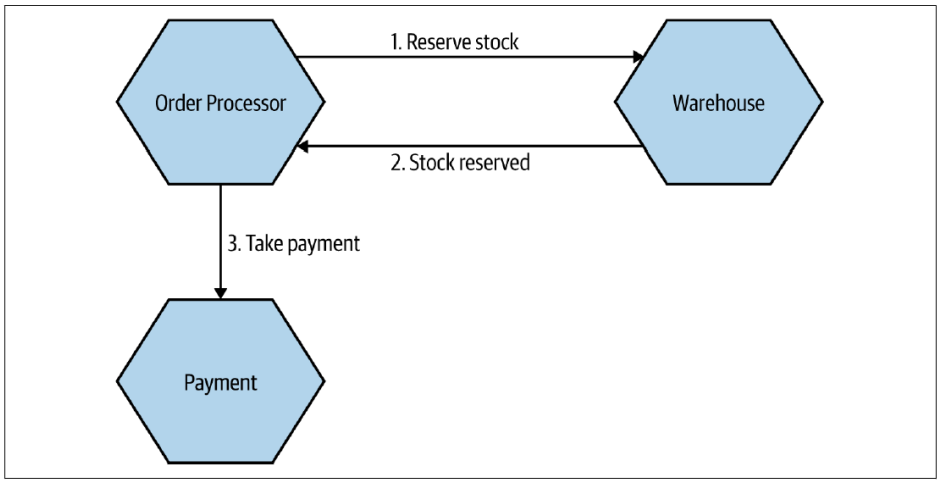
Styles de communication

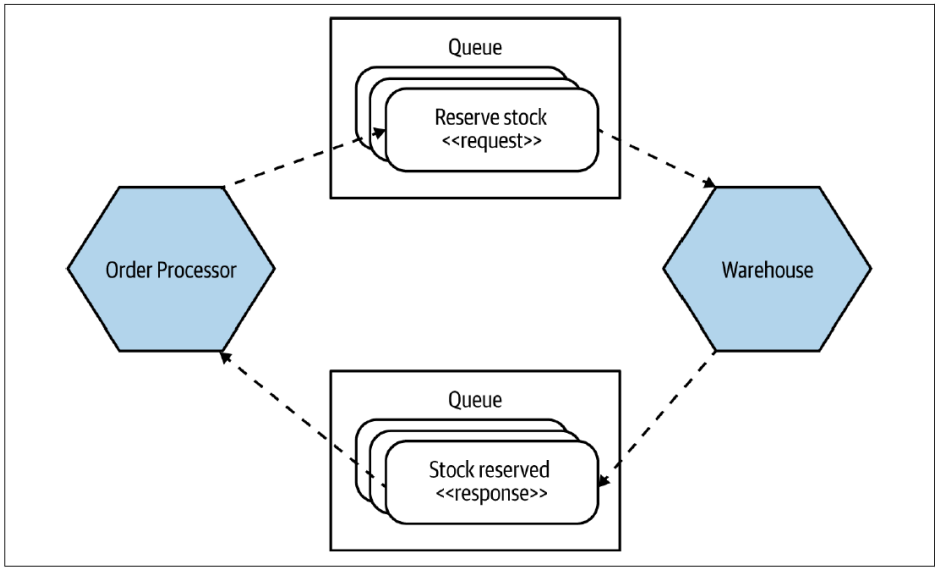


- Première question à se poser : ai-je besoin de requête/réponse ? avoir besoin d'une réponse avant le traitement suivant
- Ensuite, com sync ou com async ?
- Chacune ses avantages : simplicité vs efficacité
- Souvent, on utilise un mix de tout cela

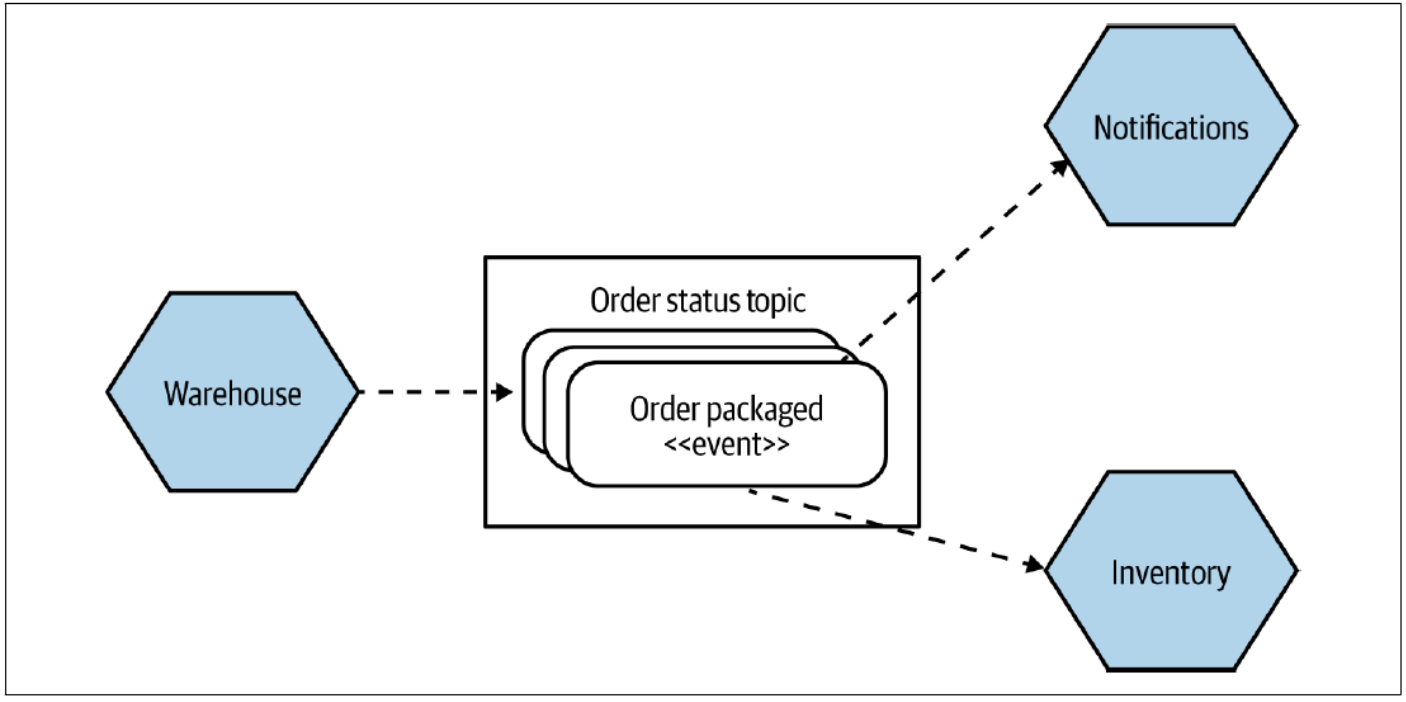
Requête/Réponse Synchrone vs asynchrone

Implémentations possibles : HTTP req/resp vs Kafka/RabbitMQ



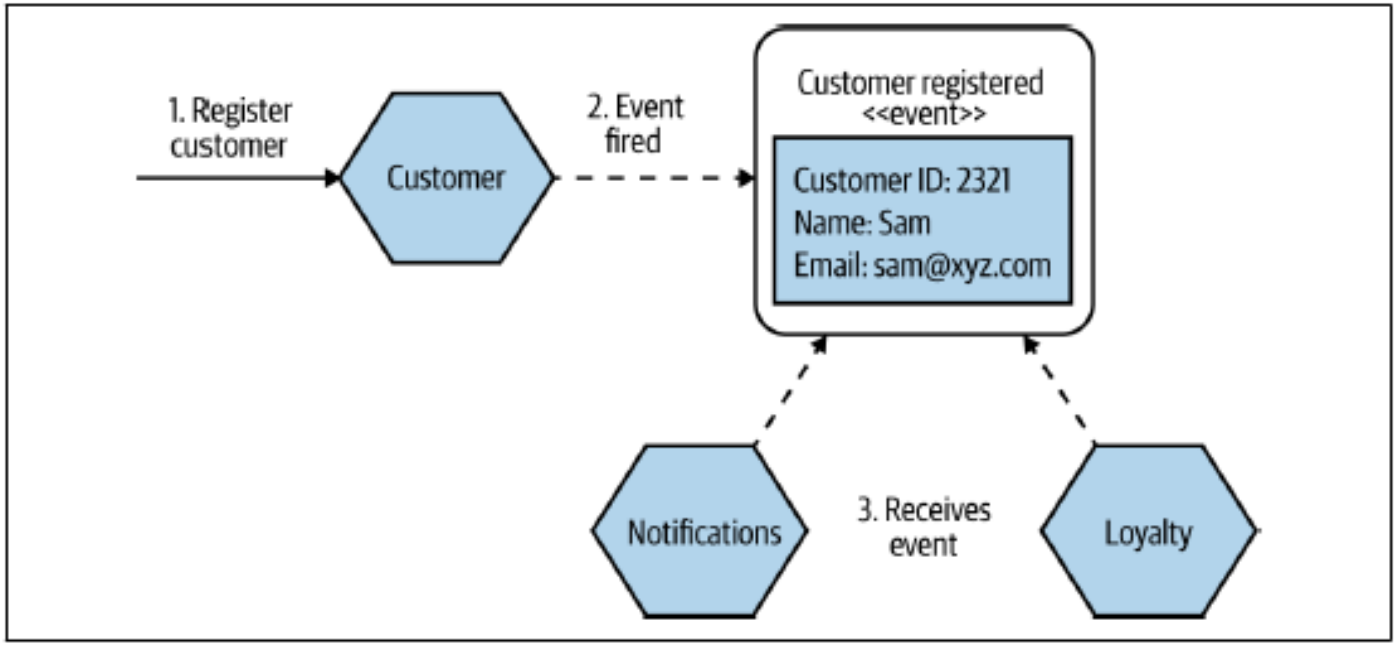


Event-driven Communication



- Couplage encore plus faible : l'émetteur ne fait aucune hypothèse sur ce que les consommateurs de l'événement vont faire de celui-ci (*responsabilité déléguée*)
- De plus en plus de devs adoptent ce mode de communication

Événements détaillés pour un couplage encore plus faible



Imaginons si l'événement ne contenait que l'ID. MS Notifications doit alors demander le nom & email du nouveau client au MS Customer (événement = contrat externe)

Points de vigilance sur la communication asynchrone

- Gestion de processus consommateurs de messages, qui sont longs à s'exécuter, pour tracer l'appelant (utiliser des "*correlation IDs*")
- Gestion des erreurs peut être compliquée : mettre en place un mécanisme de vérification de "*max retries*" en cas d'échec répété de la consommation d'un message
- Livre intéressant sur le sujet :
  Gregor Hohpe et Bobby Woolf, Enterprise Integration Patterns. Addison-Wesley, Boston 2003.

Critères de choix des implémentations

- Permettre la backward compatibility (si changement, cela ne doit pas affecter les MS clients existants)
- Rendre les interfaces explicites (schémas/ contrats explicites) et cacher les détails d'implém.
- Faire en sorte que les API soient agnostiques aux technos
- Rendre les services simples pour les clients

Implémentations possibles

- RPC (*Remote Procedure Call*), comme SOAP ou gRPC : faire des appels de fonctions locaux, qui s'exécuteront sur un MS distant
- REST : tirer profit de HTTP(S)
- GraphQL : protocole permettant d'agréger/filtrer les résultats de plusieurs requêtes côté client de MS
- Message Brokers : middlewares permettant de communiquer de façon asynchrone via des queues (1-1)/topics (1-N) – voir Apache Kafka

Implémentations possibles : RPC

- Nécessité de définir une spec de l'interface du service (un schéma) en utilisant ce que l'on appelle un *Interface Definition Language (IDL)*
- Cette spec est utilisée pour générer des stubs clients et serveurs : des objets qui rendent la distribution transparente (simple appel de fonction ou invocation de méthode)
- Certains frameworks RPC sont multi-plateformes : SOAP/WSDL et gRPC (`https://grpc.io/`)
- Faire le tutoriel gRPC :

  `https://grpc.io/docs/languages/java/quickstart/`

Implémentations possibles : RPC -suite-

- Java RMI est un autre framework RPC, mais qui contraint d'utiliser Java côté client et côté serveur
  ⇒ L'API que vous définissez sera dépendante d'une techno ↘
- Design de l'API très important : un appel de fonction local n'est pas comme un appel distant
  - Soigner la définition de paramètres (ne pas surcharger le réseau de données)
  - Ne jamais faire l'hypothèse que le réseau est fiable : gérer proprement les erreurs
  - Dans certains frameworks, il faudra re-compiler/générer les stubs à chaque modification de l'interface
- Plus de flexibilité peut être obtenu avec REST/HTTP

Implémentations possibles : GraphQL

- Utile dans le cas précis où le client fait **plusieurs** requêtes pour obtenir des données précises et ensuite agréger puis filtrer les résultats (ex : frontend dans lequel on affiche des infos sur un client et ses commandes)
- Afin d'éviter cette multitude de requêtes, on s'appuie sur GraphQL pour envoyer une seule requête
- Un MS doit exposer un endpoint GraphQL dans ce cas avec un schéma qui décrit les types disponibles
- Solution de plus en plus populaire, surtout pour la lecture de ressources à partir de frontend mobile
- A la base c'était pour JS seulement, mais désormais c'est multi-langages

Formats de sérialisation des données

- Formats spécifiques au moyen de communication : Protocol Buffer (format binaire) de Google pour gRPC (et au delà) `https://protobuf.dev/`
- Avec des brokers de messages, comme Kafka, divers formats sont possibles
- APIs REST utilisent souvent des formats textuels, comme JSON, mais le binaire est possible
  - JSON a supplanté XML pour sa simplicité
  - Sérialisation + schéma : Apache Avro (`https://avro.apache.org/`)

Recommandations sur la communication inter MS

- Rendre explicites les schémas de données est fortement recommandé : 1) réduisent la quantité de doc à écrire, et 2) préviennent les casses (structurels et sémantiques) des contrats des endpoints
  - Utiliser OpenAPI ou JSON Schema avec les API REST
- Être “lecteur tolérant” (faire un minimum d’hypothèses sur les schémas de données, qui peuvent évoluer) et respecter la loi de Postel [1] ou *robustness principle* : “*Be conservative in what you do, be liberal in what you accept from others*”

1. https://datatracker.ietf.org/doc/html/rfc761

Recommandations sur la communication inter MS -suite-

- S'appuyer sur du versioning des API (`MAJOR.MINOR.PATCH`) pour prévenir les casses (certains outils de diff de schémas sont utilisables dans une CI pour bloquer un build)
  - Exemple : Confluent Schema Registry
- Communication directe entre MS ? NON. Il faudra passer par des entités intermédiaires
  - **API Gateway**, ou de simple HTTP proxies, pour le trafic nord-sud (in backend – depuis la UI, par ex) – Spring Cloud Gateway
  - **Service Mesh** pour le trafic est-ouest (intra-backend - inter-MS) – Istio

  Ces entités doivent avoir des responsabilités très limitées (ne pas ajouter de logique métier dedans)

Sujets à débats

- Comment organiser sa base de code : monorepo ou multirepos ?
- Quel déploiement ? Machine physique ou virtuelle ? Container ou App Container/Server ? PaaS ou FaaS ?
- Différentes options de scaling : horizontal, partitionnement des données, ...
- Différentes façons de gérer le front-end : monolithe vs microfrontends