Couche de traitement

Microservices et API

Plan et objectifs

- Explorer l’architecture en microservices
- Identifier les éléments d’une API
- Identifier les caractéristiques d’une API RESTful
- Explorer les bonnes pratiques de conception d’une API RESTful
- Examiner brièvement GraphQL et l’architecture sans serveur

Qu’est-ce qu’un microservice?

Un microservice est un service autonome qui s’occupe d’un aspect précis.

Un système bâti selon une architecture en microservices contient un ensemble de plusieurs microservices distincts qui collaborent pour offrir l’ensemble des fonctionnalités du système.

Chaque service expose une API pour permettre aux différents services de communiquer entre eux.

L’API ne doit pas exposer des détails d’implémentation, mais plutôt offrir une couche d’abstraction (interface).

Loi de Conway

« les organisations qui conçoivent des systèmes [...] tendent inévitablement à produire des designs qui sont des copies de la structure de communication de leur organisation. »

— M. Conway

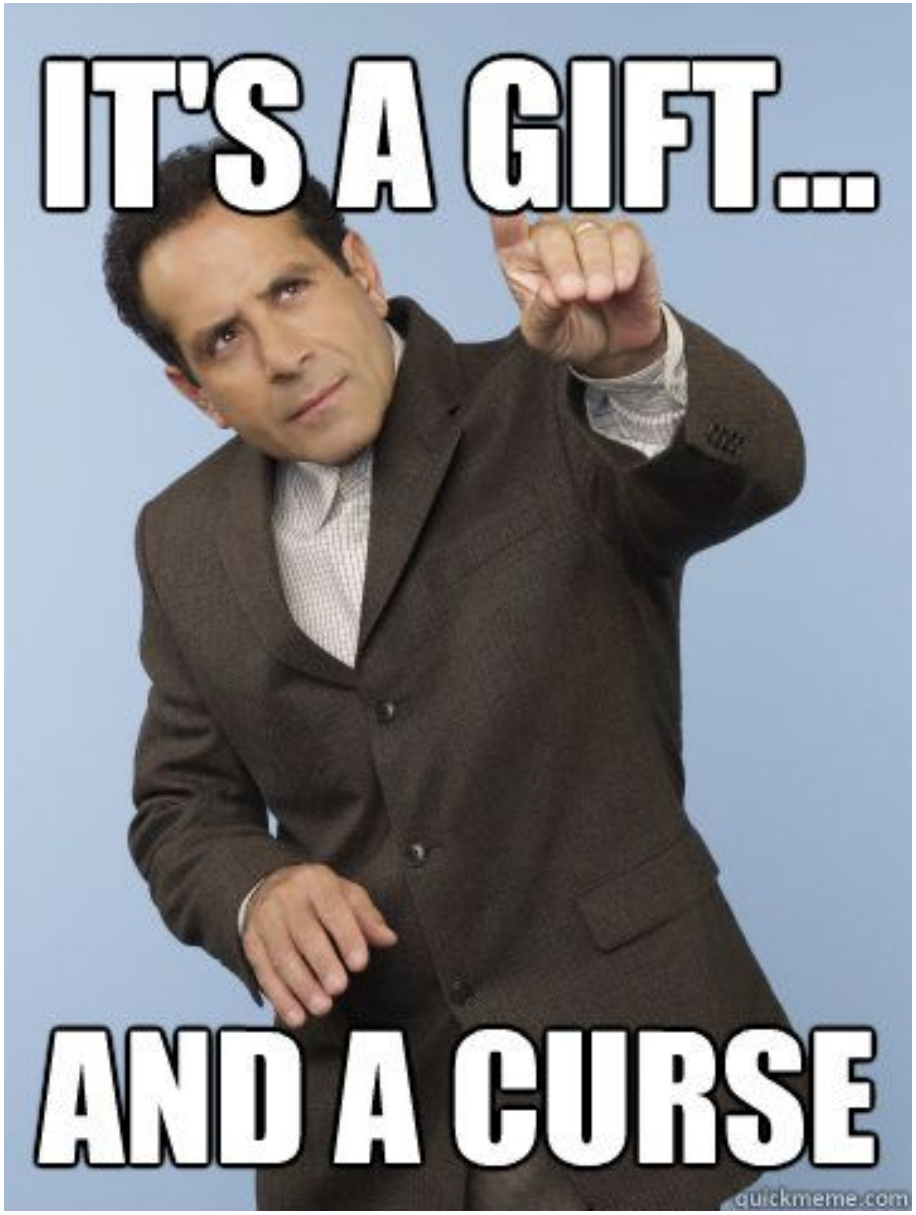
https://fr.wikipedia.org/wiki/Loi_de_Conway

Les caractéristiques d’un système par microservices

- Hétérogénéité technologique
  - il est plus simple d’utiliser ou d’expérimenter avec différentes technologies, car chaque service est indépendant
  - permet de sélectionner les technologies en fonction des requis
- Résilience
  - chaque services étant indépendant, il est possible qu’une panne n’affecte pas l’ensemble des fonctionnalités du système
- Mise à l’échelle
  - chaque service peut être mis à l’échelle indépendamment

Les caractéristiques d’un système par microservices

- Facilité de déploiement
  - chaque service peut être mis à jour indépendamment
- Composable
  - une fonctionnalité assemble plusieurs microservices
- Remplaçable
  - il est possible de remplacer (réécrire) les services indépendamment
- Aligné avec l’organisation
  - chaque équipe est responsable d’un ensemble de microservices


IT'S A GIFT...
AND A CURSE
quickmeme.com

Les caractéristiques d’un système par microservices

- Hétérogénéité technologique
  - **mais** celà augmente la charge technologique du système
  - courbe d’apprentissage élevée
- Résilience
  - **mais** lorsqu’une panne se produit, difficile d’analyser et de trouver la cause
- Mise à l’échelle
  - **mais** infrastructure plus complexe et plus difficile à gérer
  - requiert d’automatiser

Les caractéristiques d’un système par microservices

- Facilité de déploiement
  - **mais** les services doivent assurer la rétrocompatibilité
- Composable
  - **mais** difficile bien comprendre l’ensemble des options qu’offre le système, car les fonctionnalités sont morcelées
- Remplaçable
  - **mais** un plan de transition détaillé est souvent nécessaire
- Aligné avec l’organisation
  - **mais** risque de diviser le système en fonction des frontières de l’organisation et non le contraire
  - problématique de possession pour les plus vieux services

Article : Microservice Architecture at Medium

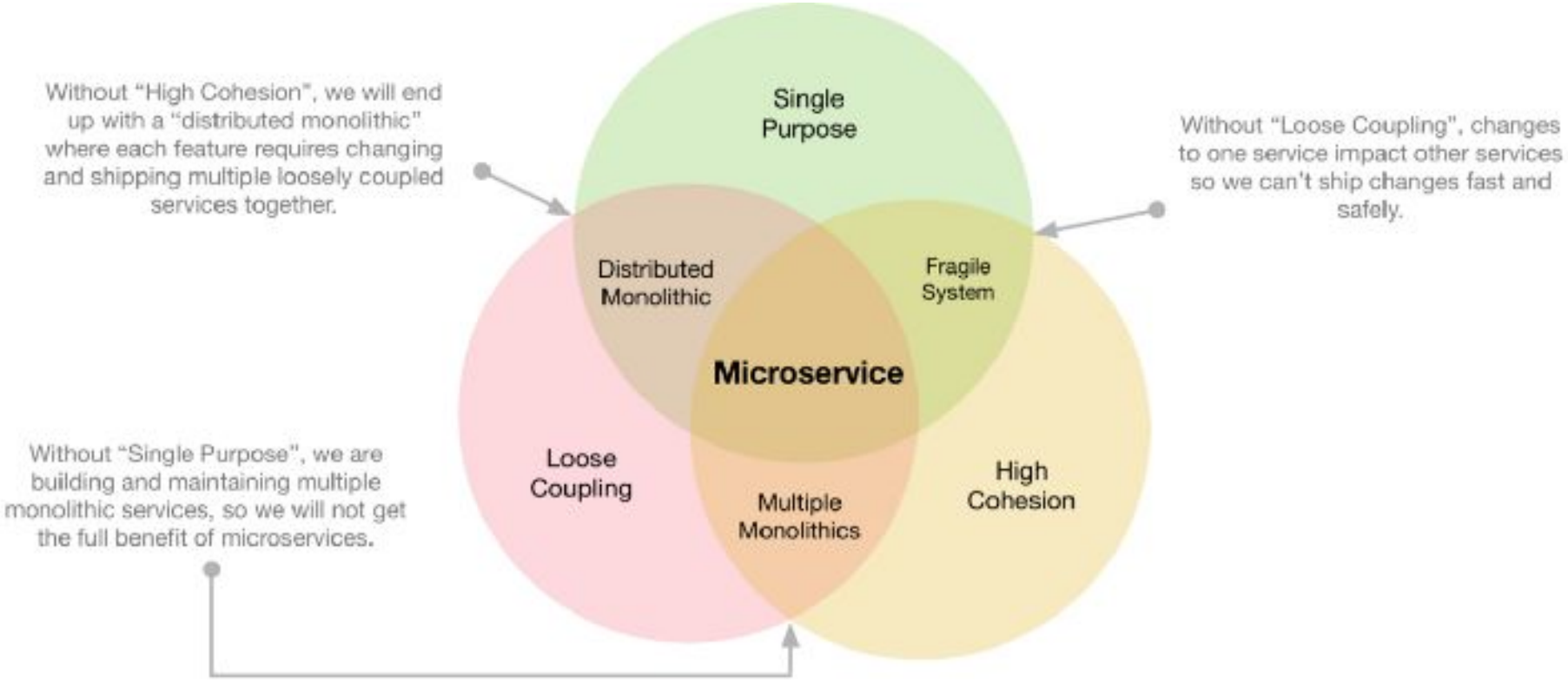
Article intéressant sur la transition de *medium.com* d'une architecture monolothique vers une architecture microservices.

Medium est une plateforme de publication de contenu type blog.

Au moment de la publication de l'article, l'auteur Xiao Ma était l'architecte en chef chez *medium*.

https://medium.engineering/microservice-architecture-at-medium-9c33805eb74f

Microservice - Principes de découpage



https://medium.engineering/microservice-architecture-at-medium-9c33805eb74f

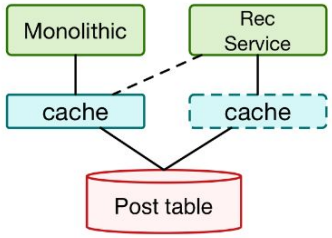
Stratégies d’adoption des microservices

- Construire des services qui ont une valeur bien définie pour le produit (fonctionnalité, ingénierie).
  - Ne pas construire pour construire.
  - Ne pas tout réécrire
- Ne pas partager le même système de persistance de données
  - Couplage entre les deux systèmes indirectement au niveau du modèle de données et du système de persistance.
- Séparer la construction d’un service et la gestion d’un service
  - La construction se concentre sur le développement de services indépendant
  - La gestion, au contraire, devrait être commune pour tous les services (réseaux, processus de déploiement, protocol, etc)
  - L’objectif est que les développeurs se concentre sur la construction et non les opérations

Stratégies d'adoption des microservices

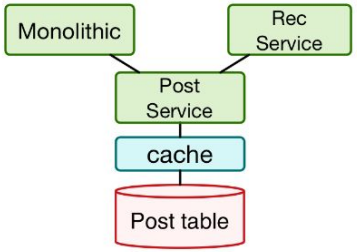
***Monolithic* Persistent Storage**

The monolithic app and the rec service share the same persistent storage.



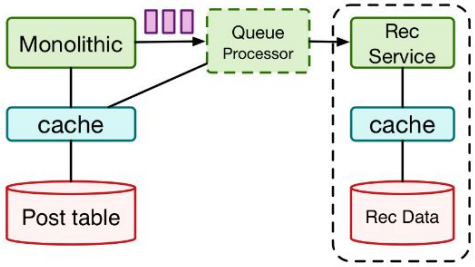
***Decoupled* Persistent Storage**

The rec service does not share the same persistent storage with the monolithic app, or any other services.



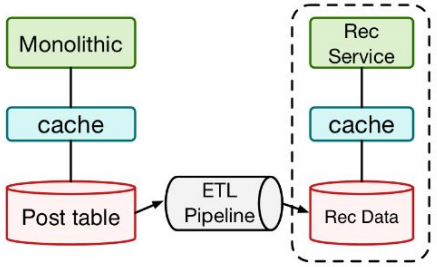
Option A

Create a Post Service that fully owns the post data. Its APIs are the only way that other services can only access post data.



Option B

Update rec service with all changes to posts that impact recommendations. Not all post changes affect recommendations.

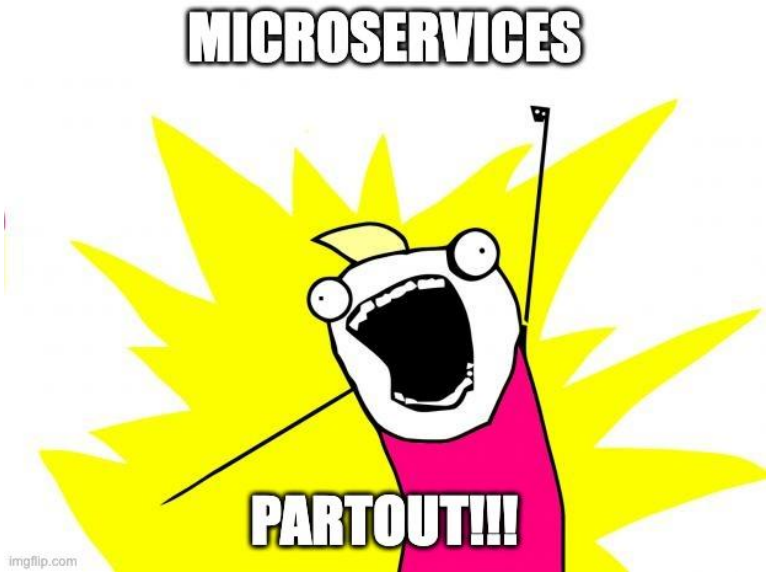


Option C

The rec service gets a non-canonical read-only copy of the post data through ELT pipelines.

Stratégies d’adoption des microservices

- Optimiser pour l’observabilité
  - Processus uniforme pour observer chacun des éléments du système pour offrir une vue cohérente
  - Standardiser les outils pour alertes, tableau de bord, logs, traces, etc
- Ne pas nécessairement partir de zéro
  - Il est possible et souhaitable de récupérer certains éléments du monolithe
- Respecter les erreurs
  - Elles vont survenir, il faut bien les gérer
- Éviter le syndrome du microservice



Doit-on éviter de construire des applications monolithiques?

Non! Souvent un bon point de départ pour valider un concept ou un produit. Plusieurs projets d'envergures reposent sur des architectures monolithiques.

Par contre, il est possible de simplifier une éventuelle migration d'architecture en

- structurant en modules le code de l'application monolithique
- en découpant en service interne les différentes fonctionnalités et accès aux données

Qu’est-ce qu’une API?

Pour que les différents services communiquent entre eux, chacun expose une API.

Une API définit l’ensemble des fonctionnalités accessibles aux clients du service.

En web, une API comporte

- un schéma de donnée (e.g. schéma JSON)
- les opérations disponibles (ajout, recherche, etc)
- le ou les mécanisme de communication utilisés pour l’utiliser (e.g. HTTP)

Une bonne API est facile à utiliser et offre aux clients les fonctionnalités dont ils ont besoin.

Le mémo *légendaire* de Jeff Bezos (traduit)

1. Toutes les équipes exposeront désormais leurs données et fonctionnalités via des interfaces de service.

2. Les équipes doivent communiquer entre elles via ces interfaces.

3. Aucune autre forme de communication interprocessus ne sera autorisée : pas de liaison directe, pas de lecture directe du magasin de données d'une autre équipe, pas de modèle de mémoire partagée, pas de porte dérobée. La seule communication autorisée est via des appels d'interface de service sur le réseau.

4. Peu importe la technologie qu'ils utilisent. HTTP, Corba, Pubsub, protocoles personnalisés - peu importe.

5. Toutes les interfaces de service, sans exception, doivent être conçues dès le départ pour être externalisables. C'est-à-dire que l'équipe doit planifier et concevoir pour pouvoir exposer l'interface aux développeurs du monde extérieur. Aucune exception.

6. Quiconque ne le fait pas sera renvoyé.

7. Merci ; Bonne journée!

API comme produit

Certaines entreprises offrent leur API en tant que produit (e.g. Stripe, Sendgrid).

Ces entreprises offres aux développeurs, des APIs spécifiques qui leur permet d'intégrer des fonctionnalités spécifique dans leur solution:

- paiement
- envoie de courriel
- automatisation de déploiement

En général, ces APIs offrent une bonne documentation et sont (parfois) des bons exemples de conception d'API.

Qui sont les clients?

Lorsqu'on conçoit une API il faut d'abord déterminer qui seront les clients et quels seront leurs besoins.

La conception de l'API peut différer selon si les clients seront des navigateurs web ou d'autres services internes.

Il faut aussi prévoir les scénarios d'utilisation pour offrir un ensemble de méthodes pertinentes et ainsi minimiser le nombre de requêtes vers l'API.

REST - RESTful

REST (REpresentational State Transfer) est un style d'architecture provenant de la dissertation de Roy Fielding. REST qui définit un ensemble de contraintes, qui, si respectées, offrent un ensemble de propriétés au système.

L'objectif était de définir une interface machine - machine évolutive, où le client, en accédant à une ressource (définie par un URI), peut déterminer automatiquement comment changer l'état de la ressource à partir de la réponse du serveur (hypermedia).

En pratique, rares sont les APIs RESTful qui respectent l'ensemble des contraintes.

API RESTful en pratique

Typiquement, un API RESTful:

- utilise HTTP(S) comme protocol de transport
- utilise les URLs comme identificateurs de ressource
- utilise les en-têtes *Accept* et *Content-Type* pour déterminer la représentation de la ressource
- utilise les verbes HTTP pour déterminer l'action à effectuer sur la ressource
- utilise les codes d'erreurs HTTP pour signifier un problème

Très populaire pour les APIs utilisés par les navigateurs web, tellement que parfois le terme *API* est confondu avec un *API de type RESTful*.

Modèle de maturité Richardson (0)

Modèle de classement d'API web basé sur 4 niveaux.

Niveau 0 - Utilisation du HTTP POST. Le contenu du post constitue la requête. Un URI par service.

URL	**Verbe**	**Opérations**
/userService	POST	recherche de comptes usager, lecture de profile, création d'usager, création de profile

Exemple de requête

```
<ProfileServiceRequest>
 <GetProfile id="1234"></GetProfile>
</ProfileServiceRequest>
```

Modèle de maturité Richardson (1)

Niveau 1 - Utilisation du HTTP POST. Le contenu du post constitue la requête. Un URI par ressource.

URL	**Verbe**	**Opérations**
/userAccount	POST	lecture, création, modification, recherche
/profil	POST	lecture, création, modification

Exemple de requête sur le profilService

```
<ProfileServiceRequest>

 <GetProfile id="1234"></GetProfile>

</ProfileServiceRequest>
```

Modèle de maturité Richardson (2)

Niveau 2 - Utilisation du verbe HTTP. Un URI par ressource.

URL	**Verbe**	**Opérations**
/profil	GET	lecture
/profil	POST	création

Modèle de maturité Richardson (3)

Niveau 3 - Utilisation du verbe HTTP. Un URI par ressource. La réponse contient des liens pour les autres opérations possibles sur la ressource.

URL	**Verbe**	**Opérations**
/profil	GET	lecture
/profil	POST	création

Exemple de réponse

{

…attributs du profile

links: { account: “http:.../account/198263” }

}

Opérations typiques - CRUD

De manière générale, il est simple de modéliser une API qui offre les opérations de création, lecture, modification et suppression d'une ressource.

URL	**Verbe HTTP**	**Opération**
/ressources	GET	Lecture d'enregistrements. Le corps de la réponse contient une liste d'enregistrements.
/ressources	POST	Création d'un nouvel enregistrement. Le corps de la requête contient les données de l'enregistrement. Typiquement, la réponse retourne l'enregistrement avec son identifiant.
/ressources/[*id*]	GET	Lecture de l'enregistrement avec l'identifiant *id*. Le corps de la réponse contient l'enregistrement.
/ressources/[*id*]	PUT	Modification de l'enregistrement avec l'identifiant *id*. Le corps de la requête contient l'enregistrement. Le corps de la réponse contient le nouvel enregistrement.
/ressources/[*id*]	DELETE	Supprimer l'enregistrement avec l'identifiant *id*.
/ressources/[id]	*PATCH*	*Modification de l'enregistrement avec l'identifiant id. Le corps de la requête contient seulement les champs à modifier. Le corps de la réponse contient le nouvel enregistrement.*

Ressources imbriquées

Il est possible d'avoir plus d'un niveau dans la hiérarchie de ressources et d'offrir plusieurs points d'entrées vers la même ressource (efficacité).

URL	Description
/livres	Les livres
/livres/[id]	Un livre en particulier
/livres/[*id*]/auteurs	Les auteurs d'un livre
/livres/[id]/auteurs/[id]	Un auteur en particulier
/auteurs/	Les auteurs
/auteurs/[id]	Un auteur en particulier

Idempotence

L'idempotence signifie qu'une opération a le même effet qu'on l'applique une ou plusieurs fois.

Dans un système distribué, où certaines opérations peuvent être ré-essayées, il est important de s’assurer que celles-ci sont idempotentes.

Selon le standard HTTP, les opérations GET, PUT, DELETE sont idempotentes.

L’opération POST ne l’est pas.

Codes HTTP

Voici les codes HTTP couramment utilisés dans une API REST et leur signification.

Code	Signification
200 OK	Indique que la requête a réussi.
201 Created	Indique que la requête a réussi et qu'une nouvelle ressource a été créée.
202 Accepted	Indique que la requête a été acceptée, mais son traitement n'est pas complété.
204 No Content	Indique que la requête a réussi, mais aucun contenu n'est retourné.

Codes HTTP

Voici les codes HTTP couramment utilisés dans une API REST et leur signification.

Code	Signification
400 Bad Request	Indique que la requête n’a pas été comprise par le serveur.
401 Unauthorized	Indique que la requête requiert de l’information d’authentification.
403 Forbidden	Indique que la requête n’est pas autorisée.
404 Not Found	Indique que le serveur ne peut trouver la ressource demandée.

Code	Signification
500 Internal Server Error	Indique que la requête a échoué à cause d’une erreur serveur.
503 Service Unavailable	Indique que le serveur ne peut répondre à la requête pour le moment.

Paramètres de requête

Il est possible de spécifier des paramètres lors d'un appel en utilisant les paramètres du URL (*query string*).

http://bibliotheque.ca/books?name=Code%20Complete&author=McConnell

Il est important de noter

- les paramètres sont nommés
- certains caractères interdits dans un URL doivent être remplacés par leur séquence d'échappement (escape sequence)
- le même paramètres peut être spécifiée plusieurs fois

Exemple: Pagination

Une méthode GET sur une ressource (sans id) est souvent une façon d'aller chercher l'ensemble des données.

En pratique, ce n'est pas toujours possible lorsque le nombre d'enregistrements récupérés est grand, car celà peut surcharger le serveur et/ou le client.

En plus de pouvoir spécifier des filtres avec les paramètres, il est souvent utile d'utiliser une méthode de pagination pour éviter que la requête retourne un nombre trop élevé d'enregistrements.

Quelques options:

- Par plage
- Par décalage
- Par page
- Par curseur

Stratégies de pagination

Par plage:

- On spécifie deux limites (début et fin) et le serveur retourne les enregistrements correspondant. (e.g. start=2022-01-01&end=2022-02-01)
- Pour parcourir la suite, on récupère la plage suivante.
- Fonctionne bien si on est certain que le nombre d'enregistrement dans une plage est limité.

Par décalage:

- On spécifie le décalage (offset) initial et un nombre d'enregistrements à récupérer. (e.g. offset=100&limit=20)
- Pour parcourir la suite, on augmente la valeur du décalage.
- Problème de duplication si des enregistrements ont été ajoutés entre deux appels.
- Nécessite d'ordonner les enregistrements.

Stratégies de pagination

Par page:

- On spécifie le nombre d'enregistrements par page et la page à récupérer. (e.g. pageSize=20&page=2)
- Pour parcourir la suite, on incrémente la valeur de la page.
- Problème de duplication si des enregistrements ont été ajoutés entre deux appels.
- Nécessite d'ordonner les enregistrements.

Par curseur:

- On identifie un champ unique dans les enregistrements (id par exemple) et on utilise l'id pour déterminer le point de départ. (e.g. startAfterId=123456&limit=20)
- Pour parcourir la suite, on utilise l'id du dernier enregistrement de la requête précédente.
- Nécessite d'ordonner les enregistrements.

Limites des APIs RESTful

Les APIs de type RESTful, se concentrent habituellement sur les opérations CRUD applicables aux différentes ressources.

Ce type d'API peut être très granulaire et nécessite beaucoup d'appels de la part des clients pour effectuer toutes les opérations nécessaires.

Aussi, si on ne fait pas attention, on repousse l'implémentation de la logique d'affaire aux clients et on se retrouve avec un service web qui n'est qu'une façade simple à la base de données (mais sans la flexibilité qu'offre la base de données pour les requêtes).

Il est aussi important de noter que de créer une transaction autour de plusieurs appels à différents services est extrêmement complexe.

Exemple: Transfert entre deux comptes

Ne pas faire à la maison!

L'API définit une ressource `account`.

Pour effectuer le transfert, le client de l'API exécute:

1. GET account/1
2. GET account/2
3. Le client valide que le transfert est possible (les fonds sont disponibles)
4. PUT account/1 avec le nouveau solde
5. PUT account/2 avec le nouveau solde

Et si la valeur du solde d'un des deux comptes changeait entre-temps?

Et si le client ou le serveur tombait en panne entre #4 et #5?

Est-ce qu'on peut ré-essayer les opérations #4 et #5 si le serveur ne répond pas?

Exemple: Transfert entre deux comptes

Les ressources:

- `/account`
- `/transfer`

Pour transférer de l'argent, on peut faire un `POST` sur `/transfer` en spécifiant le compte d'origine, le compte de destination et le montant. L'ensemble de la transaction s'effectue sur le serveur.

Une autre option serait de créer une ressource `/accountOperations` (au lieu de `transfer`) et de faire un `POST` de la même façon. Par contre, il est plus simple d'imaginer le comportement d'un `GET` sur `/accountOperations`.

Est-ce qu'on peut ré-essayer le `POST` sur `/transfer` si le serveur ne répond pas?

Comment est-ce qu'on pourrait rendre cette opération idempotente?

Gestion de versions et API

Les APIs vont évoluer.

Une bonne pratique consiste à utiliser le versionnage sémantique (voir https://semver.org/).

Lorsque que les changements ne sont pas rétrocompatibles, il faut exposer un moyen pour les clients d'utiliser la bonne version de l'API pour laquelle ils ont été conçus le temps que l'implémentation du service client se mette à jour.

Plusieurs options sont possibles:

- nouveau nom de service (si la découverte de service est utilisée)
- nouveau URL (inclusion de la version dans le URL, e.g. api.com/v2/account)
- utilisation des paramètres de requête (e.g. ?version=2)
- utilisation du type de média (e.g. Accept: application/json; version=2)
- utilisation d'une en-tête particulière (e.g. x-api-version: 2)

Exemple d’APIs

https://stripe.com/docs/api/products

https://stripe.com/docs/api/charges

https://docs.github.com/en/rest/issues/comments

Quelques guides

https://cloud.google.com/apis/design

https://learn.microsoft.com/en-us/azure/architecture/best-practices/api-design

GraphQL - https://graphql.org/

GraphQL est une alternative aux APIs RESTful qui expose les données sous forme de graphe. L'API est d'abord définie (types et fonctions) puis le client peut déterminer quels éléments de réponse il désire obtenir lorsqu'il construit sa requête.

```
# API
type Query {
  allPersons(last: Int): [Person!]!
  allPosts(last: Int): [Post!]!
}

type Person {
  id: ID!
  name: String!
  age: Int!
  posts: [Post!]!
}

type Post {
  title: String!
  author: Person!
}
```

```
# Requête
{
  allPersons {
    name
    age
    posts {
      title
    }
  }
}
```

Architecture sans serveur

Il est aussi possible d'adopter une architecture sans serveur (serverless).

Celà ne signifie pas qu'il n'y a pas de serveurs impliqués, mais que le développeur de l'application n'a pas à gérer cet aspect. Il n'a pas non plus à gérer la mise à l' échelle.

Au lieu d'avoir différents services avec des APIs distincts, le backend consiste en une suite de fonctions déployées sur une service d'infonuagique (Google Cloud Functions, AWS Lambda).

L'application web utilise ensuite ses fonctions (via appels HTTP) comme une API.

Certaines fonctions peuvent aussi être déclenchées par des événements autres (modification dans la base de données, selon un horaire établi).

CORS: Cross Origin Resource Sharing

‘’Pour des raisons de sécurité, les requêtes HTTP multi-origine émises depuis les scripts sont restreintes. Ainsi, XMLHttpRequest et l'API Fetch respectent la règle d'origine unique. Cela signifie qu'une application web qui utilise ces API peut uniquement émettre des requêtes vers la même origine que celle à partir de laquelle l'application a été chargée, sauf si des en-têtes CORS sont utilisés.”

https://developer.mozilla.org/fr/docs/Web/HTTP/CORS

Pour permettre au script d’appeler une ressource présente sur un autre domaine, le serveur doit indiquer au client quelles origines il accepte via l’en-tête `Access-Control-Allow-Origin`. On peut aussi indiquer si les cookies doivent aussi être transférés avec l’en-tête `Access-Control-Allow-Credentials`.

Inversion de control et injection de dépendance

*L'**inversion de contrôle** (inversion of control, IoC) est un patron d'architecture commun à tous les frameworks (ou cadre de développement et d'exécution). Il fonctionne selon le principe que le flot d'exécution d'un logiciel n'est plus sous le contrôle direct de l'application elle-même mais du framework ou de la couche logicielle sous-jacente.*

*L'**injection de dépendances** (dependency injection en anglais) [...] consiste à créer dynamiquement (injecter) les dépendances entre les différents objets en s'appuyant sur une description (fichier de configuration ou métadonnées) ou de manière programmatique. Ainsi les dépendances entre composants logiciels ne sont plus exprimées dans le code de manière statique mais déterminées dynamiquement à l'exécution.*

RESTful API avec Spring Boot

Spring Boot est un cadriciel java simple qui permet de bâtir des APIs RESTful simplement.

Démo…