Cloud Management Documentation

Yasser BAOUZIL

18 juillet 2025

Contexte – Jour 1

Le but du *Day 1* est de poser les fondations de notre simulateur d'infrastructure cloud minimaliste. Nous devons implémenter deux classes de base :

- `Resource` : interface abstraite pour toute entité (serveur ou conteneur) disposant de ressources CPU/RAM et pouvant démarrer/arrêter.
- `Container` : spécialisation de `Resource` représentant un conteneur nommé et doté d'une image.

Pendant cette phase, plusieurs questions de conception se sont posées :

Questions posées

1. **Pourquoi le destructeur de `Resource` est-il *protégé* et *virtuel* ?**
2. **Où et comment stocker le drapeau `active_`, et pourquoi ne pas l'exposer au constructeur ?**
3. **Pourquoi utilise-t-on `std::move(id)` pour `id` mais pas pour `cpu` ou `mem` ?**
4. **Pourquoi le membre `image_` de `Container` est-il *private* et dispose-t-il d'un accesseur, alors que les attributs de `Resource` sont *protected* sans getters ?**
5. **Qu'est-ce que le polymorphisme en programmation orientée objet ?**
6. **Quel lien entre ce polymorphisme et l'utilisation d'une classe abstraite `CloudObject` partageant `start()`/`stop()`/`getMetrics()` ?**
7. **Le destructeur de `CloudObject` doit-il être virtuel ?**
8. **Que signifient les abréviations `ctor` et `dtor`, et comment les implémenter ?**

Réponses détaillées

0.1 Destructeur `protected virtual`

— *virtual* garantit qu'au moment du `delete` d'un pointeur vers base, le destructeur de la classe dérivée est appelé, évitant les fuites et le comportement indéfini.

— *protected* interdit au code client de faire :

`Resource* r = new Server(...); delete r;`

compile error : ~Resource() inaccessible

Ce qui force l'utilisation d'un pointeur vers la classe dérivée (ou d'un `unique_ptr<Server>`), et évite les suppressions incorrectes via la base.

— Cette combinaison exprime clairement que `Resource` est une *interface abstraite* et que la destruction doit toujours passer par un type concret ou un mécanisme maîtrisé (smart pointer, factory).

0.2 Gestion de `active_`

— `active_` est un état interne (booléen) indiquant si la ressource est démarrée.

— Placer `bool active_` en *protected* permet aux sous-classes (`Server`, `Container`) d'y accéder directement sans exposer cet état au monde extérieur.

— L'initialisation est faite par défaut dans la liste d'initialisation du constructeur :

```
Resource(std::string id, double cpu, double mem)
  : id_(std::move(id)),
    cpu_(cpu),
    mem_(mem),
    active_(false)
{}
```

Cela assure toujours un état cohérent (ressource initialement arrêtée) et libère l'utilisateur de fournir ce paramètre.

— Les seules manières de modifier `active_` sont via `start()` et `stop()`, garantissant un cycle de vie maîtrisé.

0.3 `std::move(id)` vs types primitifs

— `id` est un `std::string` possédant un buffer dynamique : le déplacer (`std::move`) échange les pointeurs internes sans copier tous les caractères.

— `cpu` et `mem` sont des *primitifs* (`double`) : les copier (quelques octets) coûte moins qu'un appel à `std::move`, qui n'ajoute aucune optimisation réelle.

0.4 Encapsulation de `image_` dans `Container`

— `Resource` est une *base* destinée à plusieurs dérivées : ses attributs essentiels (`id_`, `cpu_`, `mem_`, `active_`) sont *protected* pour être réutilisables.
— `Container` est une classe *feuille* (pas d'autres dérivées) : son unique état additionnel (`image_`) reste *private* pour maximiser l'encapsulation.
— L'accesseur `getImage()` fournit au code client un accès contrôlé au nom de l'image sans exposer l'implémentation interne.

Ces choix de conception garantissent une API *sûre*, *cohérente* et *minimale*, tout en exploitant les meilleures pratiques modernes de C++.

0.5 Polymorphisme

Le polymorphisme désigne la capacité d'une même interface (méthode ou opérateur) à adopter plusieurs comportements selon le type réel de l'objet :
— *Sous-typage (runtime)* : appels dynamiques de méthodes virtuelles selon la classe concrète.
— *Paramétrique (compile)* : templates/generics permettant un même algorithme sur plusieurs types.
— *Ad-hoc (surcharge)* : même nom de fonction, signatures différentes.

0.6 Relation avec `CloudObject`

En définissant `CloudObject` comme classe abstraite (méthodes pures `start()`, `stop()`, `getMetrics()`), on fait hériter `Server` et `Container`. Grâce au polymorphisme de sous-typage, un pointeur de type `CloudObject*` peut référencer indifféremment l'un ou l'autre, et l'appel `obj->start()` ou `obj->getMetrics()` invoque la bonne implémentation à l'exécution.

0.7 Destructeur virtuel

Pour assurer qu'en faisant :

```
CloudObject* obj = new Server();
delete obj;
```

on appelle bien d'abord le destructeur de `Server` puis celui de `CloudObject`, il faut déclarer le destructeur de la base ainsi :

```
virtual ~CloudObject() = default;
```

0.8 Ctor / Dtor

- **ctor (constructor)** : méthode spéciale qui s'exécute à la création de l'objet, initialise les membres (via liste d'initialisation) et acquiert les ressources.
- **dtor (destructor)** : méthode spéciale qui s'exécute à la destruction de l'objet, libère les ressources (mémoire, fichiers, verrous...).
- *Bonnes pratiques* :
  - Utiliser systématiquement la liste d'initialisation dans le constructeur.
  - Déclarer le destructeur virtuel dans toute classe conçue pour être dérivée.