BE SOFTWARE ARCHITECTURE WISE · CHAPITRE 0

Pour commencer : comment fonctionne un logiciel

Ce chapitre est la porte d'entrée du cours. Il part du principe que vous avez déjà utilisé des applications et des sites web sans jamais programmer, il progresse vite à partir de là, et il explique depuis la base ce qu'est un logiciel, comment il représente et traite les données, comment le code est organisé, comment les ordinateurs communiquent en réseau, où les données sont stockées, comment des systèmes distincts collaborent grâce aux API, ce que sont les serveurs et le cloud, et comment les équipes construisent et mettent en production les logiciels. Les sections ultérieures abordent des idées plus difficiles comme les arbres B, les journaux d'écriture anticipée et QUIC : les débutants n'ont besoin que de l'idée principale de chacune ; ne vous souciez donc pas des détails à la première lecture. Les ingénieurs expérimentés peuvent le parcourir rapidement comme un vocabulaire commun ; tous les autres devraient le lire avant le chapitre 1, car chaque chapitre suivant s'appuie sur ces idées.

bewiseapps.com/fr/software-architecture-wise/foundations/ · Version 2026-11-03. Information pédagogique uniquement, pas un conseil professionnel.

EN UN COUP D'ŒIL

Vous apprendrez à

- Expliquer ce qui se passe, étape par étape, quand un programme s'exécute, du code source jusqu'au processeur en passant par le système d'exploitation.
- Distinguer les structures de données courantes et utiliser la notation grand O pour prévoir comment une opération ralentit quand les données augmentent.
- Décrire comment les fonctions, les modules, les objets et les interfaces découpent le code, et la différence entre une bibliothèque et un framework.
- Suivre ce qui se passe quand quelqu'un saisit une adresse web, en nommant DNS, TCP, TLS et HTTP et l'apport de chacun.
- Comparer les fichiers, les bases de données relationnelles et les principales familles NoSQL, et expliquer ce que garantit une transaction.
- Expliquer ce qu'est une API, comment fonctionne un appel REST sur HTTP et quand un message asynchrone convient mieux qu'un appel synchrone.
- Distinguer les serveurs physiques, les machines virtuelles, les conteneurs et les principaux modèles de services cloud.
- Suivre une seule action d'un utilisateur à travers toutes les couches d'un système réel, et reconnaître les décisions qui composent son architecture.

Avant de commencer

Aucun. Savoir utiliser des applications et des sites web suffit.

Durée

Environ 3 heures de lecture ; 45 minutes pour le test.

Termes clés

Processus · Compilateur et interpréteur · Structure de données · Notation grand O (Big-O) · Interface · Framework · Dépendance · Adresse IP et port · DNS · HTTP et HTTPS · Latence · Base de données et transaction · API · Conteneur et cloud

Pourquoi commencer ici

L'architecture logicielle porte sur les grandes décisions d'un système logiciel : quelles sont ses parties principales, comment elles communiquent entre elles, où vivent les données et quelles qualités, comme la rapidité, la fiabilité et la sécurité, la conception doit atteindre. Ces décisions se prennent dans un vocabulaire : processus et threads, index et caches, interfaces et dépendances, HTTP et DNS, transactions, API, conteneurs et pipelines. Les architectes emploient ces mots avec précision, et une grande partie du cours suppose de savoir ce que chacun signifie vraiment et ce qu'il coûte.

Ce chapitre fournit ce vocabulaire. Il part de quelqu'un qui n'a jamais programmé et va bien au-delà des bases à la fin, et chaque section commence par une courte analogie de la vie quotidienne avant d'expliquer le mécanisme réel. Il se veut aussi assez exact pour qu'un ingénieur expérimenté qui le lit soit d'accord avec chaque affirmation. Lorsqu'un sujet mérite beaucoup plus de profondeur, le chapitre l'introduit et renvoie à la section ultérieure qui le traite correctement.

Un programme sur un ordinateur

Les trois premières sections restent à l'intérieur d'une seule machine. Ce qu'est un logiciel explique la différence entre le matériel et le logiciel, comment le code source écrit par des personnes devient des instructions qu'un processeur peut exécuter (grâce aux compilateurs, aux interpréteurs et aux machines virtuelles), ce que fait le système d'exploitation, et pourquoi les données en mémoire disparaissent au redémarrage alors que les données sur disque sont conservées. La section se termine en suivant un programme Python de neuf lignes à chaque étape de sa vie.

Données et algorithmes montre comment les ordinateurs représentent les nombres et le texte, et présente les structures de données courantes : listes, dictionnaires, ensembles, files, piles, arbres et graphes. À l'aide de la notation grand O, elle explique pourquoi une façon de retrouver un client reste rapide avec un million de clients alors qu'une autre s'effondre, et pourquoi les index et les caches existent. Organiser le code monte ensuite des fonctions aux modules, aux objets et aux interfaces, explique la différence entre une bibliothèque et un framework, met en garde contre le coût caché des dépendances et donne un premier aperçu du couplage et de la cohésion, les deux idées qui traversent toute la conception logicielle.

Plusieurs ordinateurs qui travaillent ensemble

Les trois sections suivantes quittent la machine unique. Comment les ordinateurs communiquent explique les paquets, les adresses IP et les ports, TCP et UDP, comment le DNS transforme des noms en adresses, et comment fonctionnent les requêtes et les réponses HTTP, sécurisées par TLS. Elle souligne aussi un point qui surprend beaucoup de gens : la latence a un plancher fixé par la vitesse de la lumière, si bien qu'un aller-retour entre Londres et Sydney ne peut jamais être rapide, quelle que soit la bande passante que vous achetez.

Stocker les données couvre les fichiers et les bases de données : tables, clés et requêtes SQL, index, transactions et garanties ACID (illustrées par un virement bancaire qui ne doit jamais se faire à moitié), ainsi que les principales familles NoSQL et les cas où chacune convient. API et intégration explique comment des systèmes distincts coopèrent : l'API comme contrat, REST et

JSON sur HTTP, clés et jetons, webhooks, la différence entre attendre une réponse et envoyer un message par une file, et pourquoi les changements incompatibles d'une API pénalisent tous ceux qui en dépendent.

Où s'exécutent les logiciels et comment on les construit

Les serveurs et le cloud montre ce qu'est vraiment un serveur, comment les machines virtuelles et les conteneurs permettent à de nombreuses charges de travail de partager une même machine physique, et ce que vendent les fournisseurs de cloud : de la location de machines brutes jusqu'à l'exécution de votre code uniquement à l'arrivée d'un événement. Elle présente les régions, les zones de disponibilité, les réseaux de diffusion de contenu et les deux façons de faire face à la croissance : une machine plus grosse ou davantage de machines. Comment on construit un logiciel passe de la technologie à la pratique : exigences, gestion de versions avec Git, pyramide des tests, environnements, intégration et livraison continues, méthodes agiles et DevOps. Elle se termine en suivant un appui sur « Commander » dans une app de livraison de repas à travers chaque partie du chapitre, du téléphone au DNS, au HTTPS, à un équilibreur de charge, aux services, à une transaction en base de données, à un prestataire de paiement, à une file et à une notification push.

L'idée à retenir

Chaque système de ce chapitre est construit à partir de parties qui cachent leurs détails derrière une frontière convenue : le système d'exploitation derrière les appels système, un module derrière son interface, un serveur web derrière HTTP, une base de données derrière SQL, un prestataire de paiement derrière son API, une plateforme cloud derrière ses services. Les frontières permettent de construire de grands systèmes sans comprendre chaque détail, et c'est là que se situent les compromis importants : un appel de fonction est rapide et fiable, alors qu'un appel à travers le réseau est plus lent et peut échouer ; une dépendance épargne des efforts mais apporte des risques ; un cache fait gagner du temps mais peut servir des données périmées.

Décider où placer ces frontières, ce qui les traverse et quels compromis accepter, c'est cela, l'architecture logicielle. Chaque section se termine par une courte note sur l'importance de son sujet pour l'architecture, et le chapitre 1 commence exactement là où celui-ci s'arrête.

SECTION 0.1 SUR 0.8 · DÉBUTANT · 23 MIN DE LECTURE · DERNIÈRE RELECTURE : NOVEMBRE 2026

Programmes, code et exécution par l'ordinateur

L'IDÉE EN BREF

Un ordinateur ressemble à la cuisine d'un restaurant. Le chef (le processeur) ne sait suivre que des recettes écrites, et il les suit exactement, mot pour mot. Le plan de travail (la mémoire) contient ce que le chef utilise en ce moment et il est débarrassé chaque soir ; le garde-manger (le stockage) conserve les choses durablement. Le logiciel, c'est l'ensemble des recettes. Les gens écrivent les recettes dans une langue qu'ils savent lire, et un traducteur les convertit dans la langue très simple du chef. Le responsable de cuisine (le système d'exploitation) décide quelle recette retient l'attention du chef et évite que les postes ne se gênent.

CE QU'IL FAUT SAVOIR

- Le matériel est la machine physique (CPU, RAM, stockage, carte réseau) ; le logiciel est l'ensemble des instructions et des données qui lui disent quoi faire.
- Un programme est une liste d'instructions précises ; l'ordinateur fait exactement ce qu'elles disent, si bien que les erreurs dans les instructions deviennent des bugs.
- Les programmeurs écrivent du code source dans des langages de haut niveau, que des compilateurs, des interpréteurs ou des machines virtuelles à bytecode transforment en code machine exécutable par le CPU.
- Le système d'exploitation répartit le CPU, la mémoire, les fichiers et le réseau entre les programmes et fait respecter la sécurité ; les programmes lui demandent des services par des appels système.
- Un programme en cours d'exécution est un processus doté de sa propre mémoire ; les threads d'un processus partagent cette mémoire, ce qui est utile mais peut causer des bugs de concurrence.
- La RAM est rapide mais perd son contenu au redémarrage ; tout ce qui ne doit pas être perdu doit être écrit sur un stockage durable, en général une base de données.

Matériel et logiciel

Un ordinateur est une machine qui suit des instructions très rapidement. Les parties physiques que vous pourriez toucher forment le matériel ; les instructions qui disent à ces parties quoi faire forment le logiciel. Un ordinateur portable, un téléphone, le serveur qui fait tourner le site

web de votre banque et la puce d'un thermostat connecté sont tous des ordinateurs construits à partir des quatre mêmes types de composants.

Le CPU (unité centrale de traitement, ou processeur) exécute les instructions. Un processeur moderne possède plusieurs *cœurs*, chacun capable de traiter son propre flux d'instructions, et chaque cœur exécute des milliards d'étapes simples par seconde. La RAM (mémoire vive, généralement appelée simplement *mémoire*) contient les instructions et les données sur lesquelles le CPU travaille en ce moment. Elle est rapide, mais volatile : coupez le courant et son contenu disparaît. Le **stockage** (un disque SSD ou un disque dur à plateaux) conserve les fichiers quand l'appareil est éteint, mais il est beaucoup plus lent d'accès. Une **carte réseau** (filaire ou Wi-Fi) envoie des données vers d'autres ordinateurs et en reçoit.

Pensez à la cuisine d'un restaurant. Le chef est le CPU : le seul qui cuisine vraiment. Le plan de travail est la RAM : tout ce que le chef utilise à cet instant y est disposé, à portée de main, mais il est débarrassé en fin de soirée. Le garde-manger et le congélateur sont le stockage : vastes, permanents et à quelques pas. Le passe-plat vers la salle est la carte réseau, par lequel les commandes arrivent et les plats repartent. Les recettes sont le logiciel.

Les quatre composants de tout ordinateur

Composant	Ce qu'il fait	Analogie de la cuisine	Ce qui se passe s'il est le goulot d'étranglement
CPU (processeur)	Exécute les instructions : calculs, comparaisons, déplacement de données	Le chef	Le travail s'accumule ; les pages et les rapports deviennent lents alors qu'il reste de la mémoire libre
RAM (mémoire)	Contient les programmes en cours d'exécution et les données qu'ils utilisent	Le plan de travail	Le système commence à déplacer des données vers le disque et ralentit fortement, ou des programmes sont arrêtés
Stockage (SSD ou disque)	Conserve durablement les fichiers et les bases de données	Le garde-manger et le congélateur	L'enregistrement et le chargement des données deviennent lents ; un disque plein peut arrêter une base de données
Carte réseau	Envoie des données vers d'autres machines et en reçoit	Le passe-plat vers la salle	Les requêtes attendent d'être envoyées ou reçues ; les utilisateurs constatent des délais

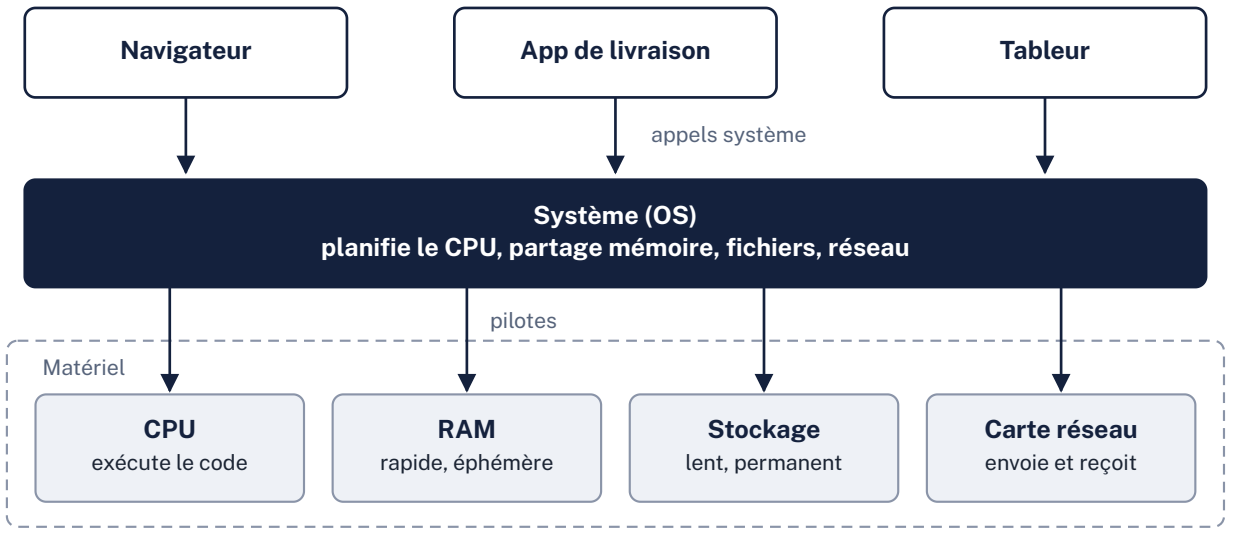
Ce qu’est un programme

Un programme est une liste d’instructions précises qu’un ordinateur suit pour accomplir une tâche : calculer une facture, afficher une page web, envoyer un paiement. Le mot qui compte est *précises*. Un cuisinier humain sait se débrouiller avec « saler à votre goût » ; un ordinateur, non. Il fait exactement ce qu’on lui dit, exactement dans l’ordre où on le lui dit, et rien de plus. Si une instruction demande de diviser par un nombre qui se révèle être zéro, l’ordinateur ne hausse pas les épaules pour continuer : le programme s’arrête sur une erreur, à moins que le programmeur n’ait écrit d’autres instructions indiquant quoi faire dans ce cas.

Au niveau le plus bas, le CPU répète un même cycle tant qu’il est allumé : aller chercher l’instruction suivante en mémoire, décoder ce qu’elle signifie, l’exécuter, puis passer à la suivante. Chaque instruction est minuscule : additionner deux nombres, comparer deux valeurs, copier une valeur d’un endroit de la mémoire à un autre, sauter à une autre instruction si une comparaison était vraie. Tout ce que fait un ordinateur, de la lecture d’un film à l’entraînement d’un modèle d’IA, est construit à partir de milliards de ces petites étapes.

Le **logiciel** est le terme le plus large : les programmes ainsi que les données, la configuration et la documentation dont ils ont besoin. Une *application* (ou app) est un logiciel qui fait quelque chose d’utile pour une personne, comme une application bancaire. Le *logiciel système*, comme le système d’exploitation décrit plus bas, existe surtout pour faire fonctionner d’autres logiciels.

Les couches : les applications, le système d’exploitation et le matériel



Les applications ne pilotent jamais directement le matériel : elles le demandent au système d’exploitation, qui le partage équitablement et en toute sécurité entre elles.

Le code source et les langages de programmation

Les CPU ne comprennent que le code machine : des instructions codées sous forme de nombres, propres à une famille de processeurs. Les puces Intel et AMD de la plupart des ordinateurs portables et des serveurs utilisent une famille (x86-64) ; les puces des téléphones, des Mac récents et de nombreux serveurs cloud en utilisent une autre (Arm). Le code machine qui tourne sur l'une ne tourne pas sur l'autre. Écrire du code machine à la main est lent, source d'erreurs et ne se fait presque plus aujourd'hui.

À la place, on écrit du code source dans un **langage de programmation** : une notation définie avec précision, qui se lit en partie comme de l'anglais et en partie comme des mathématiques. Python, JavaScript, Java, C#, Go, Rust et C en sont des exemples très répandus. On les appelle des langages de *haut niveau* parce qu'ils permettent à un programmeur de dire « additionne les prix de ce panier » en une ou deux lignes au lieu de dizaines d'instructions machine, et parce qu'ils cachent les détails du CPU utilisé. Le code source est du texte brut, enregistré dans des fichiers, et c'est ce que les programmeurs lisent, écrivent, relisent et conservent dans un système de gestion de versions (un système qui garde l'historique de chaque modification, expliqué dans la dernière section de ce chapitre).

Les langages diffèrent par ce qu'ils rendent facile. Python est apprécié pour sa lisibilité et domine la science des données et l'IA ; JavaScript est le langage qu'exécute tout navigateur web ; Java et C# sont courants dans les grands systèmes de gestion ; Go est populaire pour les services réseau ; C et Rust sont utilisés là où chaque microseconde et chaque octet comptent, comme dans les systèmes d'exploitation et les bases de données. Aucun langage n'est le meilleur en tout, et en choisir un est à la fois une décision technique et une question de compétences disponibles dans l'équipe.

Du code source à un programme en cours d'exécution : compilateurs, interpréteurs et machines virtuelles

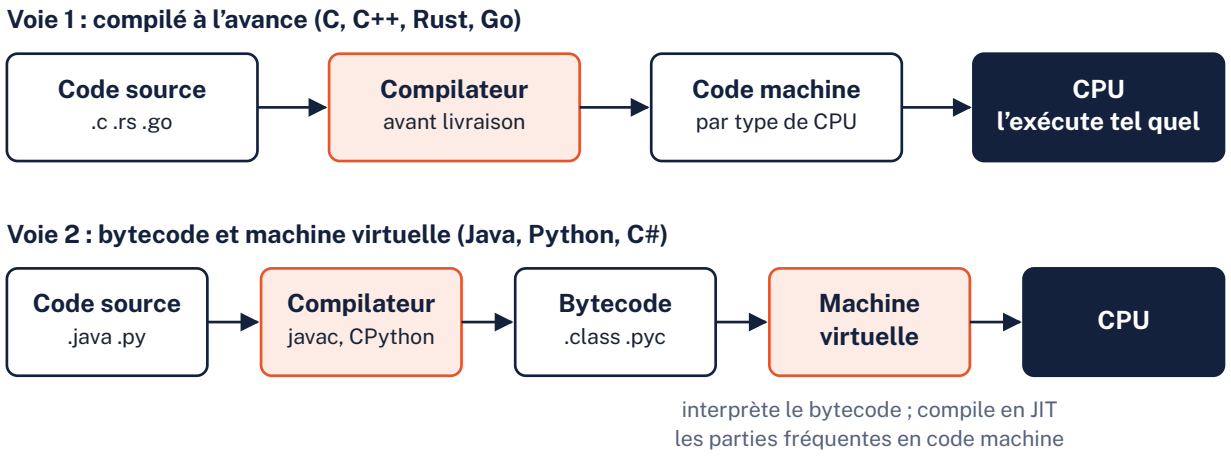
Le code source doit être transformé en quelque chose que le CPU peut exécuter. Il existe trois grandes approches, et les langages réels les combinent souvent.

Un compilateur est un programme qui traduit l'ensemble du code source d'un programme sous une autre forme avant son exécution. Des langages comme C, C++, Rust et Go sont généralement compilés *à l'avance* en code machine pour un CPU et un système d'exploitation donnés. Le résultat est un fichier *exécutable* qui démarre vite et s'exécute rapidement, mais il faut compiler une version distincte pour chaque type de machine que vous voulez prendre en charge, et le compilateur rejette le programme s'il y trouve certains types d'erreurs.

Un interpréteur est un programme qui lit le code source (ou quelque chose de très proche) et exécute ses instructions au fur et à mesure, une étape à la fois. L'interprétation permet d'essayer facilement des choses et d'exécuter le même programme sur toute machine dotée de l'interpréteur, au prix de la vitesse.

De nombreux langages populaires se situent entre les deux. Le code source Java est compilé par un outil appelé `javac` en bytecode : des instructions compactes destinées à un ordinateur imaginaire et idéalisé plutôt qu'à un vrai CPU. La **machine virtuelle Java** (JVM) exécute ensuite ce bytecode sur la machine réelle où elle est installée, ce qui permet à un même programme Java de tourner sans modification sous Windows, macOS et Linux. La JVM repère les parties du programme qui s'exécutent le plus souvent et les compile en code machine pendant l'exécution, une technique appelée compilation *juste-à-temps* (JIT). C# sur .NET fonctionne de façon similaire. L'implémentation standard de Python, CPython, compile elle aussi vos fichiers `.py` en bytecode (en mettant en cache le bytecode des modules importés dans des fichiers `.pyc`), puis exécute ce bytecode dans sa propre boucle d'interprétation.

Deux voies du code source jusqu'au CPU



La compilation à l'avance produit du code machine pour un seul type de CPU ; la compilation en bytecode permet à un même fichier de tourner partout où une machine virtuelle adaptée est installée.

ATTENTION

Deux sens de « machine virtuelle »

Les machines virtuelles de langage décrites ici (la JVM, le runtime .NET, l'interpréteur CPython) sont des programmes qui exécutent du bytecode. Elles diffèrent des machines virtuelles que louent les fournisseurs de cloud, qui imitent un ordinateur complet afin qu'un serveur physique puisse être découpé en plusieurs. Les deux s'appellent des machines virtuelles ; la seconde sorte est expliquée dans Les serveurs et le cloud.

Comparaison des approches

Approche	Langages typiques	Points forts	Coûts
Compilation à l'avance en code machine	C, C++, Rust, Go	Démarrage et exécution rapides ; un seul fichier autonome à déployer	Construire une version distincte par CPU et par système d'exploitation ; cycle modifier-essayer plus lent
Bytecode plus une machine virtuelle avec JIT	Java, Kotlin, C#	Une seule build tourne sur de nombreuses plateformes ; le JIT peut rendre très rapides les services qui tournent longtemps	Le runtime doit être installé ; démarrage plus lent et mémoire supplémentaire le temps que le JIT monte en régime
Bytecode plus un interpréteur	Python (CPython), Ruby	Rapide à écrire et à essayer ; très portable	En général plus lent pour les calculs lourds ; les parties critiques pour la vitesse sont souvent écrites en C et appelées depuis Python
Compilé en JIT dans le navigateur ou sur le serveur	JavaScript (V8 dans Chrome et Node.js)	Tourne partout où tourne un navigateur ; assez rapide pour la plupart des usages web	Les performances peuvent être moins prévisibles que celles du code compilé

Le rôle du système d'exploitation

Quand vous allumez un ordinateur, le premier programme important à démarrer est le système d'exploitation (OS) : Windows, macOS, Linux, Android ou iOS. La plupart des serveurs d'internet tournent sous Linux. Le système d'exploitation a quatre grandes missions.

Premièrement, il **partage le CPU**. Un ordinateur portable peut avoir quelques dizaines de programmes en cours d'exécution mais seulement une poignée de cœurs : l'OS fait donc passer chaque cœur d'un programme à l'autre de nombreuses fois par seconde, assez vite pour qu'ils semblent tous tourner en même temps. Deuxièmement, il **gère la mémoire**, en donnant à chaque programme sa propre zone et en empêchant un programme de lire ou d'écraser celle d'un autre. Troisièmement, il fournit **les fichiers et les périphériques** : les programmes demandent à l'OS d'ouvrir un fichier, de dessiner à l'écran ou d'envoyer des données sur le réseau, et l'OS, au moyen de petits programmes appelés *pilotes*, s'occupe du matériel particulier. Quatrièmement, il **fait respecter la sécurité** : les utilisateurs, les autorisations et la règle selon laquelle une app ne peut pas lire vos photos sans votre accord.

Les programmes font ces demandes au moyen d'*appels système*, un ensemble fixe de services proposés par l'OS. C'est le premier exemple, dans ce cours, d'une idée architecturale importante : une couche qui cache des détails compliqués derrière un ensemble stable de demandes. La personne qui écrit une app de livraison de repas n'a pas besoin de savoir de quelle marque est la puce réseau du téléphone.

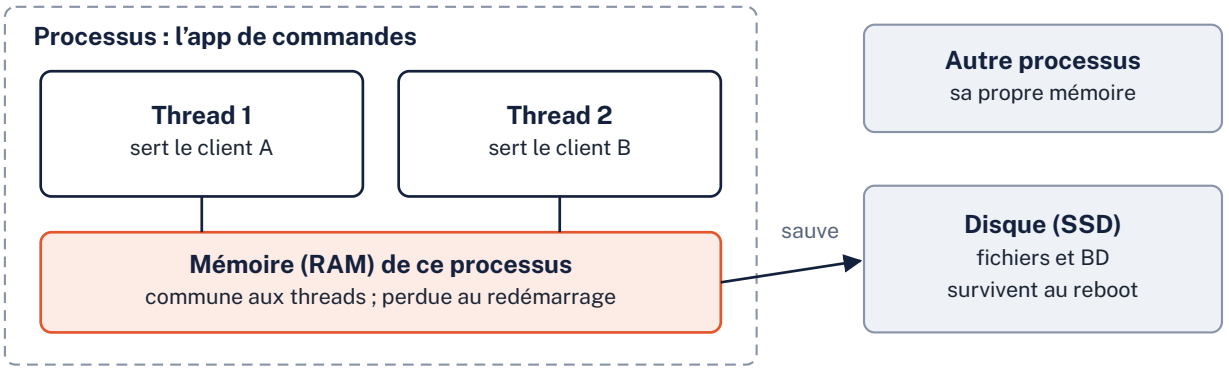
Processus, threads, mémoire et disque

Un fichier de programme posé sur le disque n'est qu'un ensemble d'instructions stockées. Quand vous le lancez, l'OS crée un processus : une instance en cours d'exécution du programme, avec sa propre mémoire privée, sa propre identité et sa propre part de temps CPU. Ouvrez deux fenêtres de navigateur et vous aurez peut-être plusieurs processus ; démarrez un serveur web et il pourra tourner sous la forme d'un seul processus ou de plusieurs.

À l'intérieur d'un processus, il peut y avoir plusieurs threads. Un thread est une séquence d'instructions en cours d'exécution. Un processus à plusieurs threads peut faire plusieurs choses à la fois, par exemple traiter en même temps les requêtes de deux clients, et tous ses threads partagent la mémoire du processus. Ce partage est pratique mais risqué : si deux threads modifient la même donnée au même moment sans coordination, le résultat peut être faux d'une manière qui n'apparaît qu'occasionnellement. Ces *bugs de concurrence* comptent parmi les plus difficiles à trouver, ce qui explique en partie pourquoi beaucoup de systèmes préfèrent des processus séparés, ou des conceptions où chaque requête est traitée indépendamment.

La différence entre la mémoire et le disque explique une grande partie de la conception logicielle. Les données en RAM sont rapides d'accès, environ un dixième de microseconde, mais elles disparaissent quand le processus se termine, plante ou que la machine redémarre. Les données sur disque survivent, mais y accéder prend bien plus longtemps : des dizaines de microsecondes ou plus sur un SSD, et plusieurs millisecondes sur un disque à plateaux. Toute information qui ne doit pas être perdue, comme une commande ou un paiement, doit donc être écrite sur un stockage durable, généralement au moyen d'une base de données, avant que le système ne dise au client « c'est fait ».

Un processus, ses threads, sa mémoire et le disque



Les threads d'un même processus partagent sa mémoire ; des processus séparés, non ; seul ce qui est enregistré sur disque survit à un redémarrage.

```
PRICE_PER_ITEM = 12.50   # pounds
VAT_RATE = 0.20          # 20 per cent

def total_with_vat(quantity):
    subtotal = quantity * PRICE_PER_ITEM
    return subtotal * (1 + VAT_RATE)

quantity = int(input("How many items? "))
print("Total to pay:", round(total_with_vat(quantity), 2))
```

Un tout petit programme Python : calculer le prix d'une commande TVA comprise

EXEMPLE COMMENTÉ

Exemple commenté : ce qui se passe quand le programme s'exécute

Supposons que le code ci-dessus soit enregistré sous le nom `order.py` et que vous tapiez `python order.py`, puis que vous saisissiez 3.

1. Démarrage	Le système d'exploitation crée un nouveau processus, lui attribue de la mémoire et lance à l'intérieur l'interpréteur Python (lui-même un programme compilé).
2. Compilation	CPython lit `order.py` sur le disque, vérifie que c'est du Python valide et le compile en bytecode. Une parenthèse ou un deux-points manquant serait signalé ici comme une *erreur de syntaxe*, avant toute exécution.
3. Définition	L'interpréteur exécute le bytecode depuis le début. Il stocke 12,5 et 0,2 en mémoire sous les noms `PRICE_PER_ITEM` et `VAT_RATE`, et enregistre la fonction `total_with_vat` sans encore l'exécuter.
4. Saisie	`input()` demande à l'OS de lire le clavier. Le processus attend, sans utiliser de CPU, jusqu'à ce que vous tapiez 3 et appuyiez sur Entrée. `int()` transforme le texte « 3 » en nombre 3.
5. Calcul	La fonction s'exécute : 3 fois 12,5 font 37,5, puis 37,5 fois 1,2 font 45,0. Chaque étape représente une poignée d'instructions de bytecode, dont chacune devient de nombreuses instructions machine sur le CPU.
6. Affichage	`print()` demande à l'OS d'afficher `Total to pay: 45.0` à l'écran.
7. Fin	Le programme atteint sa dernière ligne, le processus se termine et l'OS récupère sa mémoire. Rien n'a été enregistré sur disque : le résultat est donc perdu, sauf si vous l'avez recopié.

Même ce programme de neuf lignes dépend du système d'exploitation pour la mémoire, le clavier et l'écran, et d'un environnement d'exécution du langage pour le traduire à l'intention du CPU.

Si vous tapiez `trois` au lieu de 3, `int()` échouerait avec une *erreur d'exécution* et le programme s'arrêterait : le code ne disait jamais quoi faire d'une saisie incorrecte.

Le programme garde l'argent dans un nombre à virgule flottante pour rester court. Les vrais systèmes financiers stockent des pence entiers ou utilisent un type décimal, pour des raisons qu'explique la section suivante.

Les bugs : quand les instructions sont fausses

Un bug est une erreur dans un logiciel qui le fait se comporter autrement que prévu. Comme les ordinateurs suivent les instructions à la lettre, presque tout bug est un écart entre ce que le programmeur voulait dire et ce qu'il a réellement écrit, ou une situation à laquelle personne n'avait pensé.

Il est utile d'en distinguer trois sortes. Une **erreur de syntaxe** enfreint la grammaire du langage (une parenthèse manquante) : le compilateur ou l'interpréteur refuse alors purement et simplement d'exécuter le programme ; ces erreurs sont repérées immédiatement. Une **erreur d'exécution** survient pendant que le programme tourne, par exemple une division par zéro ou la lecture d'un fichier qui n'existe pas ; le programme s'arrête généralement avec un message d'erreur. Une **erreur de logique** est la plus dangereuse : le programme tourne sans problème et donne la mauvaise réponse, par exemple en facturant la TVA deux fois. Rien ne plante, si bien que seuls les tests, une relecture attentive ou un client mécontent la révéleront.

Les équipes professionnelles s'attendent aux bugs et mettent en place des défenses : tests automatisés, revue de code, mises en production progressives et supervision qui remarque quand les taux d'erreur augmentent. Beaucoup de ces défenses sont architecturales, et on les retrouve tout au long de ce cours.

D'OÙ CELA VIENT

La mite dans la machine

Les ingénieurs appelaient déjà les défauts des « bugs » bien avant les ordinateurs ; Thomas Edison employait le mot dans les années 1870. L'histoire la plus connue de l'informatique date du 9 septembre 1947, quand les opérateurs de l'ordinateur Harvard Mark II trouvèrent une mite coincée dans l'un de ses relais, la collèrent dans le journal de bord et écrivirent « First actual case of bug being found » (« premier cas avéré de bug trouvé »). La blague fonctionne justement parce que le terme était déjà en usage. Grace Hopper, qui travaillait sur la machine et a ensuite contribué à façonner les premiers langages de programmation, a popularisé l'histoire.

POUR ALLER PLUS LOIN

Pour aller plus loin : vous pouvez voir le bytecode

La bibliothèque standard de Python comprend un module appelé `dis` qui affiche le bytecode de n'importe quelle fonction. Exécuter `dis.dis(total_with_vat)` sur l'exemple ci-dessus liste des instructions comme `LOAD_FAST` (aller chercher une variable locale), `LOAD_GLOBAL` (aller chercher un nom défini au niveau du module), `BINARY_OP` (multiplier ou additionner) et `RETURN_VALUE`. Les instructions exactes changent d'une version de Python à l'autre, c'est pourquoi les fichiers `.pyc` sont liés à la version qui les a produits.

Pourquoi c'est important pour l'architecture

L'architecture porte sur les grandes décisions coûteuses à changer, et plusieurs d'entre elles reposent sur les idées de cette section. Le choix du langage et de l'environnement d'exécution influe sur le temps de démarrage, la consommation de mémoire, les bibliothèques utilisables et les profils que vous pouvez recruter. La séparation entre mémoire et disque détermine ce qui doit être stocké durablement, et comment un système se rétablit après un plantage. Les processus et les threads sont les unités qui s'exécutent, tombent en panne et passent à l'échelle : un chapitre ultérieur se demande si un système doit être un seul processus ou plusieurs processus qui coopèrent. Enfin, la conception en couches du système d'exploitation, qui cache le matériel derrière un ensemble stable de demandes, est la première illustration d'un principe que vous retrouverez sous le nom de masquage de l'information.

La section suivante s'intéresse à ce sur quoi travaillent réellement les programmes : les données, ainsi que les structures et les algorithmes qui rendent ce travail rapide ou lent.

VÉRIFIEZ VOTRE COMPRÉHENSION

▼ **Pourquoi les données d'un programme disparaissent-elles quand il plante, à moins d'avoir été enregistrées ?**

Pendant l'exécution d'un programme, ses données vivent en RAM, qui appartient au processus et est effacée quand le processus se termine ou que la machine redémarre. Seules les données écrites sur un stockage comme un disque ou une base de données survivent.

▼ **Quelle est la différence entre un compilateur et un interpréteur ?**

Un compilateur traduit tout le programme sous une autre forme (code machine ou bytecode) avant son exécution ; un interpréteur exécute les instructions au fur et à mesure qu'il les lit. Beaucoup de langages, comme Java et Python, compilent d'abord en bytecode, puis exécutent ce bytecode dans une machine virtuelle.

▼ **Pourquoi un programme Java construit sur un portable Windows peut-il généralement tourner sans modification sur un serveur Linux ?**

Java est compilé en bytecode, et non en code machine pour un seul CPU. Toute machine sur laquelle une machine virtuelle Java est installée peut exécuter ce bytecode.

▼ **Pourquoi les programmes demandent-ils au système d'exploitation de lire un fichier ou d'envoyer des données, au lieu de piloter eux-mêmes le disque ou la carte réseau ?**

Le système d'exploitation partage le matériel en toute sécurité entre les programmes, fait respecter les autorisations et cache les détails de chaque périphérique derrière des appels système, de sorte que les applications n'ont pas besoin de savoir sur quel matériel elles tournent.

▼ **À vous : Le programme d'une petite société de livraison garde les commandes du jour dans une liste en mémoire et les écrit dans un fichier à minuit. Le serveur perd son alimentation à 20:00. Qu'est-ce qui est perdu, pourquoi, et quel changement unique feriez-vous ?**

Toutes les commandes passées depuis la dernière sauvegarde de minuit sont perdues. La liste résidait en RAM, qui s'efface à la coupure de courant, et le fichier ne contient que les commandes de la veille. La solution consiste à écrire chaque commande sur un stockage durable, généralement une base de données, dès qu'elle est acceptée, et seulement ensuite à confirmer au client. Le coût est un peu de temps supplémentaire par commande, ce qui en vaut largement la peine.

SOURCES

- The Python Tutorial (Python documentation): https://docs.python.org/3/tutorial/index.html
- Python glossary: bytecode, interpreter and related terms: https://docs.python.org/3/glossary.html
- dis: disassembler for Python bytecode: https://docs.python.org/3/library/dis.html
- The Java Virtual Machine Specification, Java SE 21 Edition: https://docs.oracle.com/javase/specs/jvms/se21/html/index.html

SECTION 0.2 SUR 0.8 · DÉBUTANT · 20 MIN DE LECTURE · DERNIÈRE RELECTURE : NOVEMBRE 2026

Données, structures de données et algorithmes, et pourquoi la vitesse en dépend

L'IDÉE EN BREF

Imaginez que vous cherchiez une lettre dans un bureau de poste. Si les lettres sont en tas, vous devez les vérifier une par une, et un plus gros tas prend plus de temps. Si elles sont triées dans des casiers par code postal, vous allez droit au bon casier, quel que soit le nombre de lettres. Les structures de données sont les façons d'organiser les données (tas, casiers, files, piles), et les algorithmes sont les méthodes pour les traiter. Bien les choisir détermine si un système reste rapide en grandissant.

CE QU'IL FAUT SAVOIR

- Les ordinateurs stockent tout sous forme de bits ; un octet compte huit bits, et les types de données (entiers, flottants, booléens, chaînes) indiquent comment les lire.
- Les nombres à virgule flottante sont approximatifs : les montants d'argent doivent donc être stockés comme des entiers de la plus petite unité, ou avec un type décimal.
- Le texte est stocké sous forme de caractères Unicode encodés en octets, généralement en UTF-8, où un caractère peut occuper de un à quatre octets.
- Les structures de données comme les listes, les dictionnaires, les ensembles, les files, les piles, les arbres et les graphes rendent chacune différentes opérations rapides.
- La notation grand O décrit comment le travail d'un algorithme augmente avec les données : $O(1)$, $O(\log n)$, $O(n)$, $O(n \log n)$, $O(n^2)$.
- Les index et les caches échangent de l'espace supplémentaire (et le travail de les tenir à jour) contre des réponses plus rapides.

Tout n'est que bits

Les ordinateurs stockent tout sous forme de bits : des chiffres binaires qui valent soit 0, soit 1. Physiquement, un bit peut être une minuscule charge électrique, une zone magnétique sur un disque ou une impulsion lumineuse dans une fibre, mais pour le logiciel ce n'est qu'une valeur parmi deux. Huit bits forment un octet, qui peut contenir $2^8 = 256$ motifs différents, de 00000000 à 11111111.

Les tailles se construisent à partir de là. Un kilooctet vaut environ mille octets, un mégaoctet environ un million, un gigaoctet environ un milliard et un téraoctet environ mille milliards. (En toute rigueur, les préfixes standard kilo, méga et giga désignent des puissances de 1000 ; la mémoire se mesure souvent en puissances de 1024, pour lesquelles existent les unités kibioctet, mébioctet et gibioctet. La différence compte rarement pour la conception, mais elle explique pourquoi un disque de « 1 To » affiche moins dans certains gestionnaires de fichiers.) Une page de texte brut fait quelques kilooctets, une photo prise au téléphone quelques mégaoctets, une heure de vidéo haute définition quelques gigaoctets.

Les bits ne signifient rien à eux seuls. Le motif 01000001 peut être le nombre 65, la lettre A, une partie d'une couleur dans une photo ou une partie d'une instruction pour le CPU. Ce qui donne un sens aux bits, c'est la convention sur la façon de les lire : c'est l'idée de type de données.

Les types de données : nombres, texte, vrai ou faux

Un type de données indique à un programme quel genre de valeur est une donnée, et donc quelles opérations ont un sens. Tout langage de programmation possède quelques types de base, et ceux de Python sont représentatifs.

Les **entiers** sont des nombres sans partie décimale, comme 3, 0 ou -42, qui servent à compter : les articles d'un panier, les sièges d'un vol. Les **nombres à virgule flottante** (flottants), comme 12,5 ou 0,2, représentent les nombres décimaux de façon approchée, en binaire. Comme beaucoup de nombres décimaux n'ont pas de forme binaire exacte, `0.1 + 0.2` donne `0.30000000000000004` en Python. Cela convient à la physique et aux graphismes, mais pas à l'argent : c'est pourquoi les systèmes financiers stockent les montants en nombres entiers de pence ou de centimes, ou utilisent un type décimal (Python dispose de `decimal.Decimal`). Les **booléens** n'ont que deux valeurs, `True` et `False`, et enregistrent des faits de type oui ou non, comme « cette commande a-t-elle été payée ? ». Les **chaînes** sont du texte, comme le nom d'un client. Les systèmes réels ajoutent d'autres types par-dessus : dates et heures, montants monétaires et données binaires comme les images.

Les types comptent au-delà d'un seul programme. Quand deux systèmes échangent des données, ils doivent s'accorder sur les types autant que sur les valeurs : un code postal est-il un nombre ou du texte (du texte : certains commencent par des lettres, et certains codes numériques commencent par zéro) ? Une date est-elle au format jour-mois ou mois-jour, et dans quel fuseau horaire ? Beaucoup de bugs d'intégration coûteux sont des désaccords sur les types.

Le texte et UTF-8

Le texte est stocké en attribuant un numéro à chaque caractère, puis en stockant ces numéros sous forme d'octets. La norme internationale **Unicode** attribue un numéro, appelé point de code, à bien plus de cent mille caractères issus des écritures du monde entier, plus des symboles et des emoji. Un *encodage* décide ensuite comment ces numéros deviennent des octets. Aujourd'hui, l'encodage dominant, sur le web et dans la plupart des nouveaux systèmes, est UTF-8.

UTF-8 utilise un nombre variable d'octets par caractère : un octet pour les lettres de base de l'anglais, les chiffres et la ponctuation (si bien qu'un texte qui n'utilise que ces caractères est identique, octet pour octet, en UTF-8 et dans l'encodage ASCII, bien plus ancien), deux octets pour des caractères comme `é`, trois pour des caractères comme `€` et la plupart des caractères chinois, japonais et coréens, et quatre pour les emoji et certaines écritures plus rares. Cela a une conséquence pratique : le nombre de caractères d'une chaîne et le nombre d'octets qu'elle occupe peuvent différer. Une colonne de base de données limitée à 50 octets ne pourra peut-être pas contenir un nom de 50 caractères écrit en grec.

Le bug de texte classique, ce sont les caractères illisibles (par exemple `CafÃ©` au lieu de `Café`), qui apparaissent quand des octets écrits dans un encodage sont lus dans un autre. Le remède est simple à énoncer : utiliser UTF-8 partout, et le dire explicitement chaque fois que du texte franchit une frontière, comme un fichier, une base de données ou un message réseau.

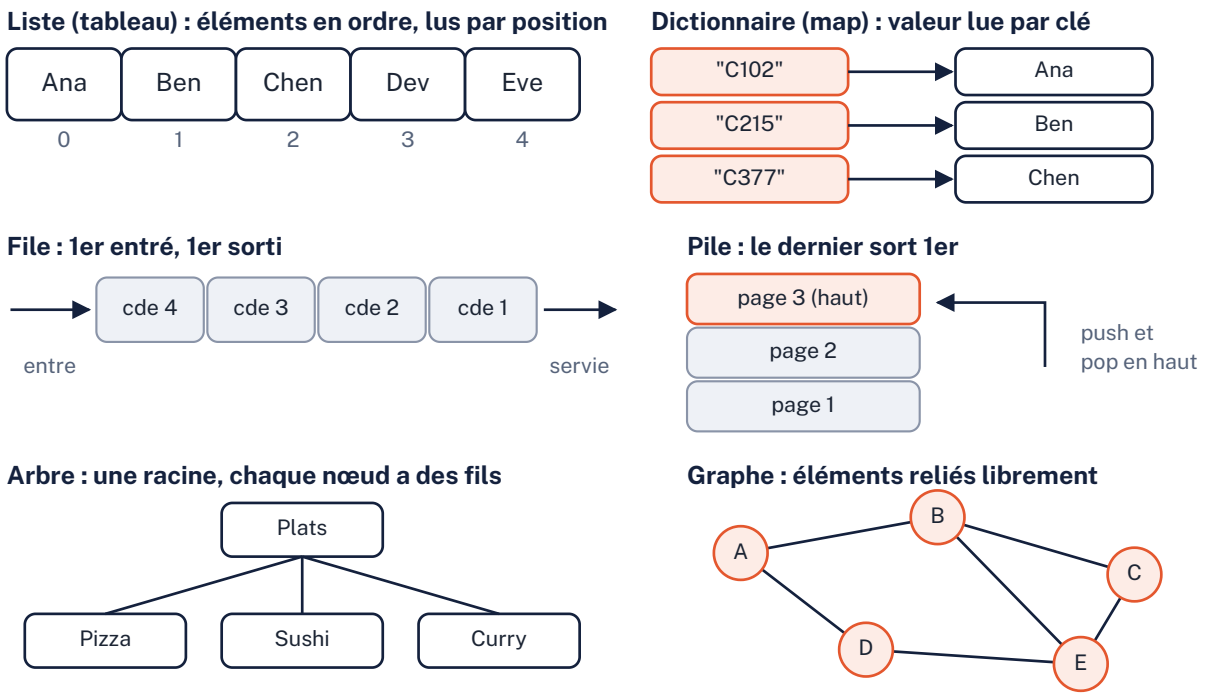
Les structures de données : des façons d'organiser les données

Les programmes traitent rarement une seule valeur à la fois. Une structure de données est une façon d'organiser de nombreuses valeurs en mémoire pour que les opérations dont vous avez besoin soient rapides. Le bon choix peut faire la différence entre une page qui se charge instantanément et une page qui ne se charge jamais. Chacune a son analogie dans la vie quotidienne.

Une **liste** (appelée *tableau* dans beaucoup de langages) garde les éléments dans l'ordre, comme une liste de courses numérotée. Obtenir le cinquième élément est instantané, car l'ordinateur peut calculer où il se trouve. Trouver un élément d'après sa valeur oblige à vérifier les éléments un par un. Un dictionnaire (appelé aussi *map* ou *table de hachage*) range des valeurs sous des clés uniques, comme un vestiaire : vous tendez le ticket C377 et vous récupérez aussitôt le bon manteau, sans regarder tous les manteaux. Un **ensemble** est une collection sans doublons, comme une liste d'invités, et répond rapidement à la question « cette personne y figure-t-elle ? ».

Une **file** fonctionne comme une file d'attente au guichet de la poste : premier entré, premier sorti. Les systèmes utilisent des files pour le travail en attente, comme les commandes qui attendent la cuisine. Une **pile** fonctionne comme une pile d'assiettes : dernier entré, premier sorti. Le bouton Précédent de votre navigateur en utilise une : la page que vous avez visitée le plus récemment est la première à laquelle vous revenez. Un **arbre** a une seule racine avec des branches en dessous, comme un arbre généalogique ou les dossiers de votre ordinateur ; c'est aussi grâce aux arbres que les bases de données gardent les données triées pour des recherches rapides. Un **graphe** est un ensemble d'éléments (nœuds) reliés par des liens (arêtes) selon n'importe quel schéma, comme une carte routière ou un réseau social, et répond à des questions comme « quel est le plus court chemin ? » ou « qui sont les amis de mes amis ? ».

Six structures de données courantes



Chaque structure organise le même genre de données différemment, ce qui rend différentes questions rapides.

Ce pour quoi chaque structure est efficace

Structure	Rapide pour	Lente pour	Usage typique
Liste / tableau	Obtenir l'élément situé à une position ; ajouter à la fin	Trouver un élément d'après sa valeur (parcours obligatoire) ; insérer près du début	Un panier d'articles ordonné ; une page de résultats de recherche
Dictionnaire / map	Trouver, ajouter ou supprimer d'après la clé	Tout ce qui exige les éléments dans un ordre trié	Clients par identifiant ; paramètres par nom ; un cache
Ensemble	Tester l'appartenance ; supprimer les doublons	Conserver un ordre (la plupart des ensembles n'en ont pas)	Utilisateurs ayant déjà voté ; étiquettes uniques
File	Ajouter à l'arrière et prendre à l'avant	Atteindre les éléments du milieu	Commandes qui attendent la cuisine ; tâches qui attendent un worker
Pile	Ajouter et prendre au sommet	Atteindre les éléments plus bas	Historique d'annulation ; le bouton Précédent ; suivi des appels de fonctions
Arbre (trié)	Trouver d'après la valeur et lister une plage dans l'ordre	Plus complexe à maintenir équilibré	Index de base de données ; dossiers ; menus
Graphe	Suivre des relations	Les très grands graphes exigent un stockage spécialisé	Itinéraires et cartes ; réseaux sociaux ; recommandations

```
basket = ["pizza", "cola", "pizza"]            # list: ordered, duplicates allowed
print(basket[0])                                # by position: "pizza"

customers = {"C102": "Ana", "C377": "Chen"}     # dictionary: key to value
print(customers["C377"])                        # by key: "Chen"

voted = {"ana@example.com", "ben@example.com"}  # set: no duplicates
print("ana@example.com" in voted)               # True

from collections import deque
kitchen = deque(["order 1", "order 2"])         # queue
kitchen.append("order 3")                       # joins at the back
print(kitchen.popleft())                        # served first: "order 1"
```

Les mêmes structures en Python

Les algorithmes

Un algorithme est une méthode précise, étape par étape, pour résoudre un problème, et qui se termine toujours : comme une recette, mais sans aucune place pour l'interprétation. Trier une liste de noms, trouver le plus court chemin entre deux adresses, vérifier si un mot de passe correspond et décider quelle publicité afficher, tout cela est fait par des algorithmes.

Il existe généralement plus d'un algorithme pour un même problème, et la quantité de travail qu'ils font peut varier énormément. Imaginez que vous cherchiez un nom dans un annuaire téléphonique imprimé. Vous pourriez commencer à la première page et lire chaque entrée jusqu'à le trouver : c'est correct, mais lent. Ou vous pourriez ouvrir l'annuaire au milieu, regarder si le nom vient avant ou après cette page, écarter la moitié qui ne peut pas le contenir, et recommencer. Comme l'annuaire est trié, chaque coup d'œil divise par deux les pages restantes. Cette seconde méthode s'appelle la **recherche dichotomique**, et pour un annuaire d'un million de noms, elle demande environ 20 coups d'œil au lieu de jusqu'à un million.

La recherche dichotomique : diviser le problème par deux



En surbrillance : l'élément du milieu comparé à chaque étape. En gris : écarté. Restants : 16, 7, 3, 1

Dans une liste triée, chaque comparaison écarte la moitié de ce qui reste : 16 éléments demandent donc au plus 4 ou 5 coups d'œil.

Grand O : comment le travail augmente avec les données

Les ingénieurs décrivent comment le coût d'un algorithme augmente à l'aide de la notation grand O. Elle ignore le temps exact sur une machine particulière et pose une seule question : si la quantité de données, appelée *n*, devient dix ou mille fois plus grande, combien de travail en plus faut-il ? Au sens strict, c'est une borne supérieure de la croissance du travail, et on la cite généralement pour le pire cas, sauf mention contraire. Elle ignore les facteurs constants, ce qui fait à la fois sa force (elle saisit la forme de la croissance) et sa faiblesse (deux algorithmes en O(n) peuvent quand même différer d'un facteur dix en pratique).

O(1), temps constant : le travail ne dépend pas de n. Chercher une clé dans un dictionnaire est en O(1) en moyenne. **O(log n)**, logarithmique : le travail augmente d'une étape chaque fois que n double, comme la recherche dichotomique. **O(n)**, linéaire : deux fois plus de données, deux fois plus de travail, comme le parcours d'une liste. **O(n log n)** : un peu moins bien que linéaire ; les bons algorithmes de tri généralistes fonctionnent à ce rythme. **O(n^2)**, quadratique : deux fois plus de données, quatre fois plus de travail. Cela apparaît typiquement quand, pour chaque élément, un programme parcourt tous les autres éléments, par exemple pour comparer chaque client à tous les autres afin de trouver les doublons.

Nombre approximatif d'étapes de base quand les données augmentent

n (éléments)	O(1)	O(log n)	O(n)	O(n log n)	O(n^2)
10	1	environ 3	10	environ 33	100
1,000	1	environ 10	1,000	environ 10 000	1,000,000
1,000,000	1	environ 20	1,000,000	environ 20 000 000	1,000,000,000,000

Lisez attentivement la dernière colonne. À un milliard d'étapes simples par seconde, un algorithme en million au carré a besoin de mille milliards d'étapes : environ un quart d'heure de calcul pur, et en pratique bien plus. Le même travail fait avec une méthode en O(n log n) prend une petite fraction de seconde. Un code qui semblait correct pendant les tests avec 1000 clients peut s'enliser complètement avec un million, et aucune quantité de matériel supplémentaire ne sauvera longtemps un algorithme quadratique.

EXEMPLE COMMENTÉ

Exemple commenté : retrouver un client parmi un million

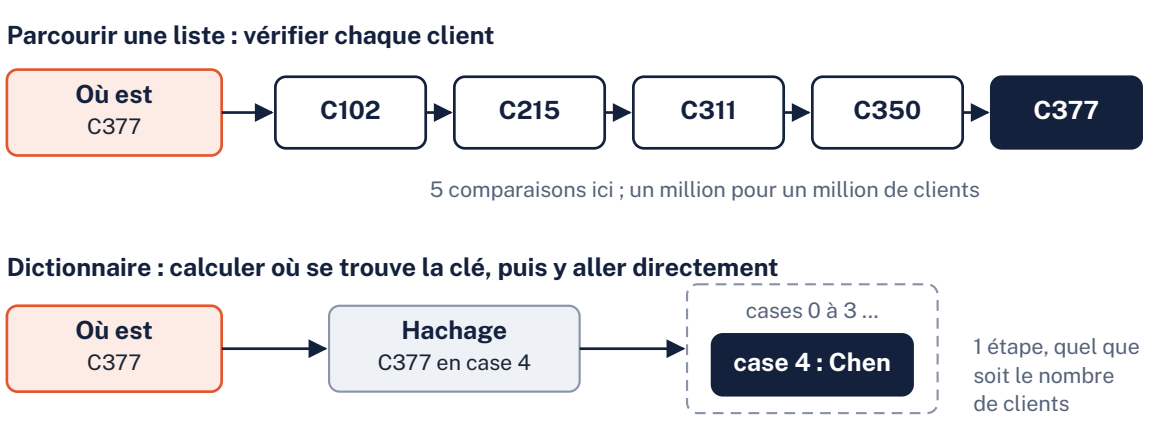
Une app de livraison de repas stocke 1 000 000 de clients. Chaque fois qu'une commande arrive, elle doit retrouver le client d'après son identifiant. Comparez deux conceptions, en supposant que chaque vérification prend 0,1 microseconde.

Conception A : une liste	`for c in customers: if c.id == wanted: ...` parcourt la liste jusqu'à trouver une correspondance.
Vérifications par recherche (A)	Jusqu'à 1 000 000 ; en moyenne environ 500 000.
Temps par recherche (A)	Environ 500 000 x 0,1 microseconde = 50 millisecondes.
À 200 commandes par seconde (A)	200 x 50 ms = 10 secondes de travail CPU chaque seconde : le serveur ne peut pas suivre, alors les requêtes s'accumulent et expirent.
Conception B : un dictionnaire indexé par identifiant	`customers_by_id[wanted]` calcule où se trouve l'entrée et y va directement.
Vérifications par recherche (B)	Environ 1, qu'il y ait mille clients ou un million.
Temps par recherche (B)	Bien moins d'une microseconde.
Coût de la conception B	De la mémoire supplémentaire pour le dictionnaire, et le travail de le construire puis de le tenir à jour chaque fois qu'un client est ajouté ou modifié.

Les deux conceptions donnent des réponses identiques ; seule la structure de données diffère, et cela fait passer le résultat de l'échec à une marge confortable.

Les bases de données offrent le même choix. Sans index, trouver une ligne d'après une colonne oblige à parcourir toute la table ; avec un index, il suffit de quelques étapes dans un arbre trié. La section sur le stockage des données y reviendra.

Parcours ou accès direct



Un parcours fait un travail proportionnel au nombre d'éléments ; un dictionnaire va directement à la bonne case.

L'espace contre le temps

Le dictionnaire de l'exemple commenté est plus rapide parce qu'il utilise de la mémoire supplémentaire pour organiser les données à l'avance. C'est l'un des compromis les plus courants en informatique : on peut souvent acheter de la vitesse avec de l'espace, ou économiser de l'espace en dépensant du temps.

Un index est un ensemble de données supplémentaires, conservé à côté des données principales, qui répond rapidement à un certain type de question, comme l'index à la fin d'un livre. Il accélère les lectures, mais il occupe de l'espace de stockage et doit être mis à jour à chaque écriture : une table dotée de dix index est donc lente pour les insertions. Un cache conserve une copie de la réponse à une question coûteuse dans un endroit rapide, par exemple en mémoire, pour que la personne suivante qui la pose l'obtienne immédiatement. Les caches expliquent pourquoi la page d'un produit populaire se charge vite même quand la base de données derrière est très sollicitée, mais ils soulèvent une question difficile : quand les données d'origine changent, à quelle vitesse la copie doit-elle changer aussi ? Afficher un prix périmé pendant quelques secondes peut être acceptable ; afficher un solde bancaire périmé ne l'est peut-être pas.

La compression est l'échange inverse : dépenser du temps CPU pour tasser les données dans moins d'octets, ce qui économise du stockage et du temps de transfert réseau.

ATTENTION

Idées reçues courantes

- **« La notation grand O indique à quelle vitesse le code s'exécute. »** Elle indique comment le coût augmente. Pour dix éléments, une méthode en $O(n^2)$ peut être plus rapide qu'une méthode astucieuse en O(n log n) ; la notation grand O compte quand n devient grand.
- **« Un matériel plus rapide corrige les algorithmes lents. »** Une machine deux fois plus rapide ne permet à un algorithme quadratique de traiter qu'environ 1,4 fois plus de données dans le même temps.
- **« Les recherches dans un dictionnaire sont toujours instantanées. »** Elles sont en O(1) en moyenne, mais utilisent plus de mémoire qu'une liste et n'offrent aucun ordre : « tous les clients dont le nom commence par C » exige donc toujours un parcours ou une structure triée.
- **« Les flottants conviennent pour l'argent. »** Les erreurs d'arrondi s'accumulent sur des millions de transactions. Utilisez des entiers de la plus petite unité, ou un type décimal.

POUR ALLER PLUS LOIN

Pour aller plus loin : moyennes, pires cas et hachage

Un dictionnaire atteint O(1) en moyenne en faisant passer chaque clé par une *fonction de hachage*, qui transforme la clé en un nombre désignant une case. Deux clés peuvent tomber dans la même case (une *collision*) ; les implémentations gèrent ce cas, et dans une situation pathologique où beaucoup de clés entrent en collision, les recherches se dégradent jusqu'à O(n). Ce n'est pas que théorique : des attaquants ont envoyé à des serveurs web des paramètres de requête conçus pour entrer en collision, c'est pourquoi les langages modernes, dont Python, rendent aléatoire le hachage de leurs chaînes. Les index de base de données utilisent généralement un arbre équilibré appelé B-tree (arbre B), qui offre des recherches en O(log n) et, contrairement à une table de hachage, prend aussi en charge les plages et l'ordre trié.

Pourquoi c'est important pour l'architecture

Les architectes écrivent rarement des algorithmes de tri, mais ils prennent sans cesse des décisions dont les conséquences sont exactement celles-ci. À quelles questions le système devra-t-il répondre rapidement, et les données ont-elles les index nécessaires pour y répondre ? Jusqu'où les données auront-elles grossi dans trois ans, et quelles parties de la

conception sont linéaires, ou pire, face à cette croissance ? Où la mise en cache vaut-elle sa complexité, et à quel point les données en cache peuvent-elles être périmées ? Quand le volume de données est multiplié par dix, quel composant casse en premier ?

Ce sont des questions de performances et de scalabilité, que le chapitre 2 précise, ainsi que de mise en cache et de choix de base de données, que le chapitre 7 traite en profondeur. La section suivante passe des données au code, et à la façon d'organiser le code pour que des personnes puissent le comprendre et le modifier.

VÉRIFIEZ VOTRE COMPRÉHENSION

▼ Pourquoi `0.1 + 0.2` ne donne-t-il pas exactement 0,3 dans la plupart des langages ?

Les nombres à virgule flottante sont stockés en binaire, et beaucoup de fractions décimales, dont 0,1 et 0,2, n'ont pas de forme binaire exacte : de minuscules erreurs d'arrondi apparaissent donc.

▼ Quelle structure de données convient à un bouton « Annuler » dans un éditeur de texte, et pourquoi ?

Une pile : chaque action est empilée au sommet, et l'annulation dépile d'abord la plus récente (dernier entré, premier sorti).

▼ Si un traitement en O(n) prend une seconde pour 100 000 éléments, combien de temps environ faut-il pour 1 000 000 ?

Environ dix secondes, car un travail linéaire augmente proportionnellement aux données.

▼ Que payez-vous en échange de la vitesse que vous donne un index de base de données ?

Du stockage supplémentaire, et du travail supplémentaire à chaque insertion, mise à jour et suppression pour tenir l'index à jour.

▼ À vous : Une app de livraison de repas parcourt une simple liste de 2 000 000 clients pour en trouver un par son identifiant, et chaque vérification prend 0,1 microseconde. À 50 commandes par seconde, un seul cœur de processeur peut-il suivre ? Que changeriez-vous ?

En moyenne, une recherche effectue 1 000 000 de vérifications, soit 100 millisecondes. À 50 commandes par seconde, cela fait 5 secondes de travail à chaque seconde : un cœur prend donc du retard et les commandes s'accumulent. Utilisez un dictionnaire ou un index de base de données indexé sur l'ID client, afin que chaque recherche prenne à peu près le même court temps, quel que soit le nombre de clients. Vous payez en mémoire supplémentaire et en travail de mise à jour.

SOURCES

- Python tutorial: data structures: https://docs.python.org/3/tutorial/datastructures.html
- Python wiki: time complexity of built-in operations: https://wiki.python.org/moin/TimeComplexity
- Unicode HOWTO (Python documentation): https://docs.python.org/3/howto/unicode.html
- RFC 3629: UTF-8, a transformation format of ISO 10646: https://www.rfc-editor.org/rfc/rfc3629

SECTION 0.3 SUR 0.8 · DÉBUTANT · 19 MIN DE LECTURE · DERNIÈRE RELECTURE : NOVEMBRE 2026

Fonctions, modules, objets, interfaces, bibliothèques et frameworks

L'IDÉE EN BREF

Revenez au restaurant de la section 0.1, mais regardez maintenant à l'intérieur d'un programme plutôt que l'ordinateur entier. Au lieu d'un seul cuisinier qui fait tout, il y a des postes (grill, pâtisserie, salades), chacun avec son personnel, ses outils et ses recettes. Les commandes passent d'un poste à l'autre par un passe-plat avec un système de bons, si bien que le grill n'a pas besoin de savoir comment travaille le pâtissier. Le code s'organise de la même façon : de petites tâches nommées (les fonctions) sont regroupées en postes (les modules), et les postes ne communiquent entre eux que par des bons convenus (les interfaces). Ainsi, un poste peut changer ses méthodes sans perturber le reste de la cuisine.

CE QU'IL FAUT SAVOIR

- Une fonction est un morceau de code nommé et réutilisable, avec des entrées et une sortie ; elle permet d'écrire une règle une seule fois et de la modifier à un seul endroit.
- Les modules et les paquets regroupent du code apparenté et séparent une petite partie publique de détails privés qui peuvent changer librement.
- Les classes réunissent des données et les méthodes qui agissent sur elles ; l'encapsulation garde les données privées, si bien que les règles sont toujours respectées.
- Une interface est un contrat entre des parties ; un code qui dépend d'une interface plutôt que d'une implémentation peut facilement changer d'implémentation.
- Vous appelez une bibliothèque ; un framework appelle votre code. Cette inversion de contrôle rend les frameworks puissants, mais plus difficiles à remplacer.
- Les dépendances, récupérées par des gestionnaires de paquets comme pip et npm, font gagner des efforts, mais apportent du code transitif, des risques de sécurité et du travail de maintenance.
- Une bonne structure, c'est un couplage faible et une forte cohésion : le changement reste peu coûteux et les équipes peuvent travailler de façon indépendante.

Pourquoi organiser le code ?

Un petit script de vingt lignes se lit de haut en bas en une minute. Un vrai système, comme une plateforme de livraison de repas, peut compter des centaines de milliers, voire des millions de lignes, écrites par des dizaines de personnes pendant de nombreuses années. Personne ne

peut garder tout cela en tête. La seule façon de travailler sur un tel système est de le découper en parties que l'on peut comprendre, tester et modifier une à une, avec des règles claires sur la manière dont les parties peuvent s'utiliser entre elles.

Les langages de programmation nous offrent pour cela une échelle de briques, de la plus petite à la plus grande : les fonctions, les modules et les paquets, les classes et les objets, et les interfaces entre eux. Au-dessus du code que nous écrivons nous-mêmes se trouvent les bibliothèques et les frameworks écrits par d'autres. Cette section gravit cette échelle. Ces mêmes idées, appliquées à plus grande échelle, sont l'objet même de l'architecture logicielle.

Les fonctions : nommer une tâche, puis la réutiliser

Une fonction est un morceau de code nommé qui fait une seule tâche. Elle reçoit des *entrées* (appelées paramètres ou arguments), effectue un travail et renvoie généralement une *sortie* (sa valeur de retour). Pensez à une machine à café : vous mettez des grains, de l'eau et vous choisissez une boisson ; elle fait son travail ; un café en sort. Vous n'avez pas besoin de savoir comment fonctionne la pompe pour vous en servir.

Les fonctions apportent trois avantages. **La réutilisation** : le calcul de la TVA est écrit une seule fois et appelé depuis le paiement, la facture et l'écran de remboursement. **La lisibilité** : `total_with_vat(basket)` dit ce qui se passe sans montrer comment. **Le changement à un seul endroit** : quand le taux de TVA change, une seule fonction change, et non vingt copies du même calcul éparpillées dans le code (des copies qui finiraient vite par diverger).

Une bonne fonction fait une seule chose, porte un nom qui dit ce qu'elle fait, reçoit en paramètres les entrées dont elle a besoin au lieu d'aller chercher des données cachées, et est assez courte pour se lire d'un coup d'œil. Une fonction qui se contente de calculer sa sortie à partir de ses entrées, sans rien modifier d'autre, est appelée fonction *pure* ; les fonctions pures sont particulièrement faciles à tester, car les mêmes entrées donnent toujours la même sortie.

```
VAT_RATE = 0.20

def total_with_vat(prices_in_pence):
    """Return the total of a basket in pence, including VAT."""
    subtotal = sum(prices_in_pence)
    return round(subtotal * (1 + VAT_RATE))

checkout_total = total_with_vat([1250, 399])   # 1979
invoice_total = total_with_vat([5000])         # 6000
```

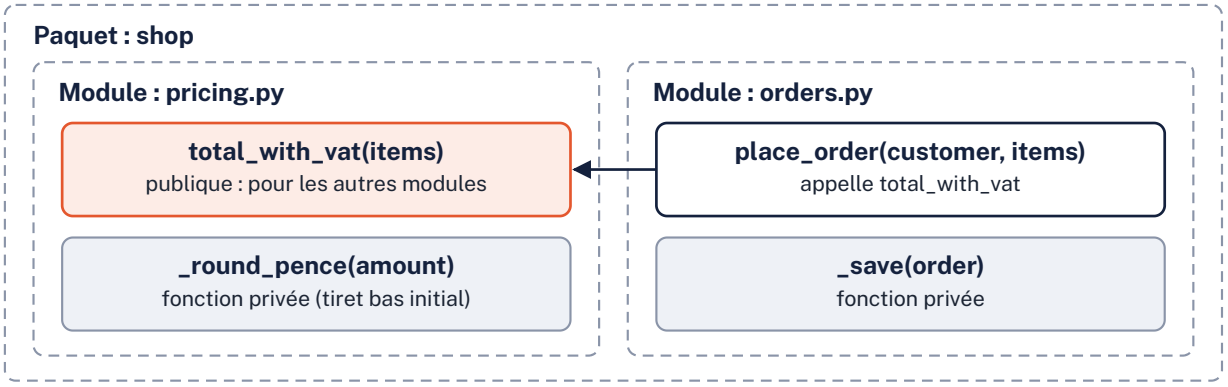
Une seule fonction, réutilisée à plusieurs endroits

Les modules et les paquets : regrouper le code apparenté

Un module regroupe des fonctions et des données apparentées dans une unité qui porte son propre nom. En Python, un module est simplement un fichier `.py` ; `pricing.py` pourrait contenir tout ce qui concerne les prix, les remises et les taxes. Un autre fichier l'utilise en écrivant `import pricing`. Un **paquet** (package) est un dossier de modules apparentés : une boutique pourrait ainsi avoir un paquet `shop` contenant les modules `pricing`, `orders` et `customers`. D'autres langages emploient d'autres mots (Java a des packages, C# des espaces de noms et des assemblies, JavaScript des modules), mais l'idée est la même.

Un module ressemble à un service dans une organisation. Le service financier a un guichet public où les autres peuvent faire des demandes (« merci d'émettre cette facture »), et des procédures internes que personne à l'extérieur n'a besoin de voir. De la même façon, les modules séparent une petite partie **publique**, destinée aux autres modules, de détails **privés**. Python signale les noms privés par convention avec un tiret bas initial, comme `_round_pence` ; Java et C# imposent la confidentialité avec des mots-clés tels que `private`. L'intérêt de cette séparation est que la partie privée peut être réécrite librement sans rien casser ailleurs, tant que la partie publique tient ses promesses.

Fonctions, modules et paquet



Fonctions dans des modules ; modules liés réunis en paquet ; seuls les noms publics servent à l'extérieur

Le module orders n'utilise que la fonction publique du module pricing, jamais sa fonction privée.

Les classes et les objets : des données avec leur comportement

Beaucoup de langages permettent de réunir des données et les fonctions qui travaillent dessus. Une **classe** est le modèle ; un **objet** est un exemplaire fabriqué à partir de ce modèle. Une classe `BankAccount` pourrait indiquer que chaque compte a un titulaire et un solde, et peut accepter des dépôts et des retraits. Le compte de chaque client est alors un objet distinct créé à partir de cette classe. Les fonctions qui appartiennent à une classe s'appellent des *méthodes*. (Voir l'entrée du glossaire classe et objet.)

L'idée la plus utile ici pour l'architecture est l'encapsulation : un objet garde ses données privées et ne laisse le reste du code les modifier qu'au moyen de ses méthodes. Si n'importe qui pouvait modifier directement `account.balance`, rien n'empêcherait du code situé ailleurs de rendre un solde négatif ou de sauter le journal d'audit. Avec l'encapsulation, la méthode `withdraw` est la seule porte d'entrée, et elle peut faire respecter les règles à chaque fois. L'analogie est celle du guichetier de banque : les clients ne peuvent pas entrer dans la chambre forte, mais ils peuvent s'adresser au guichetier, qui vérifie les règles avant de remettre l'argent.

La programmation orientée objet va bien au-delà (héritage, polymorphisme, patrons de conception), et les praticiens ne s'accordent pas sur ce qui en est vraiment utile : certains langages, comme Go, se passent volontairement de l'héritage, et beaucoup d'équipes préfèrent un style plus fonctionnel. L'encapsulation, en revanche, est précieuse quel que soit le style.

```
class BankAccount:
    def __init__(self, owner):
        self.owner = owner
        self._balance = 0          # private by convention

    def deposit(self, pence):
        if pence ⩽ 0:
            raise ValueError("Deposit must be positive")
        self._balance += pence

    def withdraw(self, pence):
        if pence ⩽ 0:
            raise ValueError("Withdrawal must be positive")
        if pence > self._balance:
            raise ValueError("Insufficient funds")
        self._balance -= pence

    def balance(self):
        return self._balance

acct = BankAccount("Ana")
acct.deposit(10000)
acct.withdraw(2500)
print(acct.balance())   # 7500
```

L'encapsulation : le solde ne peut changer que par des méthodes qui font respecter les règles

Les interfaces : des contrats entre les parties

Une interface est l'ensemble convenu des opérations qu'une partie d'un système offre aux autres : leurs noms, les entrées qu'elles acceptent, ce qu'elles renvoient et ce qui peut mal tourner. C'est un contrat. L'appelant s'appuie sur le contrat, pas sur la manière dont le travail est

fait. Une prise électrique est une interface : tout appareil muni de la bonne fiche fonctionne, et peu importe à l'appareil que l'électricité provienne du vent, du gaz ou du nucléaire.

Dans le code, une interface permet de remplacer une implémentation par une autre. Supposons qu'une boutique accepte aujourd'hui les paiements par carte via un prestataire de paiement, mais qu'elle puisse en changer l'an prochain. Si le code de paiement appelle directement, partout, les fonctions propres au prestataire, changer de prestataire impose de réécrire le paiement. Si, au contraire, la boutique définit sa propre petite interface `PaymentGateway` avec une seule opération, `charge(amount, card_token)`, puis écrit un adaptateur par prestataire, le code de paiement ne dépend que de l'interface, et changer de prestataire revient à écrire un nouvel adaptateur. La même astuce facilite les tests : dans les tests, une fausse passerelle qui ne contacte jamais de vraie banque peut remplacer la vraie.

Pour les architectes, les interfaces sont l'idée la plus importante de cette section. Chaque frontière d'un système, de la signature d'une fonction à une API web utilisée par des milliers d'autres entreprises, est une interface. Les bonnes interfaces sont petites, stables et cachent les décisions susceptibles de changer ; ce principe, appelé masquage de l'information, a été formulé par David Parnas en 1972 et il est traité au chapitre 3.

```
from typing import Protocol

class PaymentGateway(Protocol):
    def charge(self, amount_pence: int, card_token: str) → bool: ...

class ProviderAGateway:
    def charge(self, amount_pence, card_token):
        ...  # call Provider A's web API
        return True

class FakeGateway:                      # used in tests
    def charge(self, amount_pence, card_token):
        return amount_pence < 100_000   # pretend large payments fail

def checkout(basket_total, card_token, gateway: PaymentGateway):
    return "paid" if gateway.charge(basket_total, card_token) else "declined"
```

Une interface (ici un Protocol Python) avec deux implémentations interchangeables

Les bibliothèques et les frameworks : du code écrit par d'autres

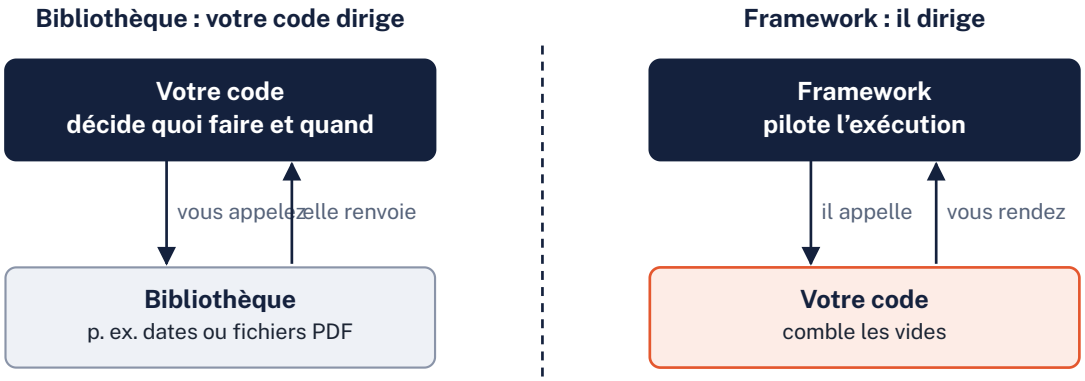
Presque personne n'écrit tout à partir de zéro. Une bibliothèque est du code réutilisable que quelqu'un d'autre a écrit et que votre code appelle quand il en a besoin : une bibliothèque pour gérer les dates et les fuseaux horaires, pour générer des PDF, pour dialoguer avec une base de

données. Vous gardez la main ; la bibliothèque est un outil que vous prenez et que vous reposez.

Un framework, c'est différent. Il fournit le squelette d'ensemble d'une application et décide du moment où les choses se produisent ; vous comblez les vides avec votre propre code, et le framework appelle ce code aux bons moments. Un framework web comme Django (Python), Spring (Java), Ruby on Rails ou Express (JavaScript) reçoit chaque requête web entrante, détermine laquelle de vos fonctions doit la traiter, appelle cette fonction et transforme son résultat en page web ou en données. Le moyen le plus simple de retenir la différence : *vous appelez une bibliothèque ; un framework appelle votre code*.

Ce renversement s'appelle l'inversion de contrôle. Martin Fowler attribue à Ralph Johnson et Brian Foote l'emploi de ce terme en 1988, et l'idée est souvent résumée par le principe d'Hollywood : «Ne nous appelez pas, c'est nous qui vous appellerons». Une franchise de restauration est une analogie assez juste : le franchiseur fournit le bâtiment, le système de commande, les horaires d'ouverture et les règles ; le franchisé fournit la cuisine dans les créneaux que le franchiseur lui laisse.

Qui appelle qui



Le sens du contrôle est la vraie différence entre une bibliothèque et un framework.

Bibliothèque ou framework

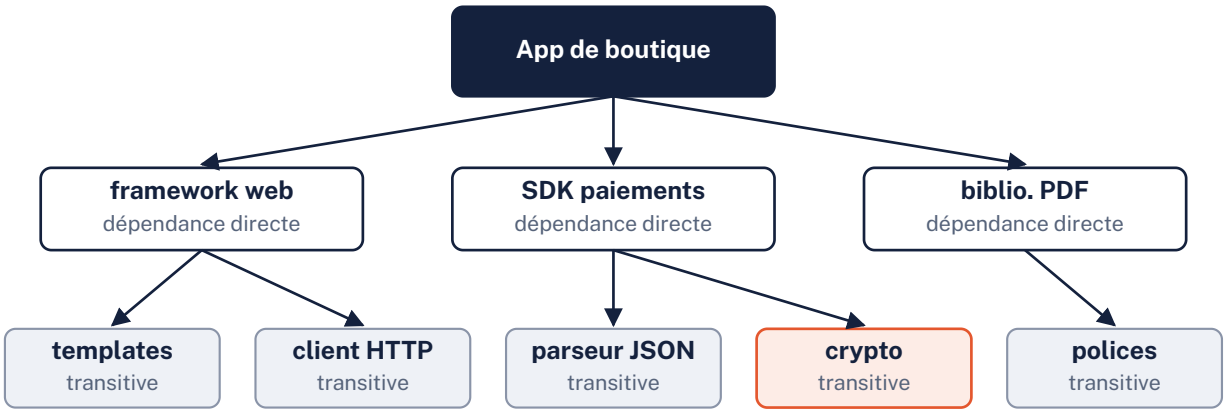
	Bibliothèque	Framework
Qui contrôle le déroulement	Votre code	Le framework (inversion de contrôle)
Exemples	Une bibliothèque de dates, un générateur de PDF, un parseur JSON	Django, Spring, Ruby on Rails, Angular
Facilité de remplacement ultérieur	En général moyenne : modifier les endroits qui l'appellent	En général difficile : la structure du framework façonne toute l'application
Principal avantage	Évite d'écrire soi-même une fonctionnalité	Évite de concevoir toute la structure ; des conventions communes dans l'équipe
Principal risque	Une dépendance de plus à maintenir	Dépendance à sa façon de travailler et à son cycle de mises à jour

Les dépendances et les gestionnaires de paquets

Chaque bibliothèque ou framework dont votre code a besoin est une dépendance. Les projets modernes en ont beaucoup ; ils utilisent donc un gestionnaire de paquets pour les récupérer : `pip` pour Python (depuis le Python Package Index, PyPI) et `npm` pour JavaScript (depuis le registre npm) sont les plus connus, avec des équivalents comme Maven pour Java et NuGet pour .NET. Un projet liste ce dont il a besoin, souvent avec des plages de versions, et le gestionnaire de paquets télécharge ces paquets et, surtout, les paquets dont *eux-mêmes* dépendent. Ces dépendances indirectes, dites *transitives*, sont souvent bien plus nombreuses que les dépendances directes.

Les versions sont couramment numérotées MAJEUR.MINEUR.CORRECTIF selon une convention appelée versionnage sémantique : un changement du numéro majeur signale une modification incompatible, un changement du numéro mineur de nouvelles fonctionnalités qui ne devraient pas gêner les utilisateurs existants, et un changement du numéro de correctif des corrections de bugs. Les projets conservent aussi un *fichier de verrouillage* (lock file, comme `package-lock.json` pour npm) qui enregistre la version exacte de chaque paquet installé, afin que la machine de chaque développeur et le build de production utilisent le même code.

Dépendances directes et transitives



Vous avez choisi trois paquets ; vous dépendez de huit. Une faille dans l'un d'eux (p. ex. crypto) touche aussi votre app.

La plus grande partie du code exécuté par votre application découle des choix de quelqu'un d'autre.

Les dépendances font gagner énormément d'efforts, mais chacune est du code que vous n'avez pas écrit, exécuté avec toutes les permissions du vôtre. Deux incidents bien connus montrent les risques. En mars 2016, un développeur a retiré ses paquets de npm à la suite d'un différend, parmi lesquels `left-pad`, une minuscule fonction d'environ onze lignes servant à compléter du texte ; des milliers de projets qui en dépendaient, directement ou indirectement, ont soudain cessé de compiler jusqu'à ce que npm le rétablisse. En décembre 2021, une grave faille de sécurité, surnommée Log4Shell, a été révélée dans Log4j 2, une bibliothèque de journalisation Java très utilisée ; des organisations du monde entier ont passé des semaines à retrouver chaque système qui la contenait, souvent enfouie plusieurs niveaux plus bas dans leurs dépendances.

Liste de contrôle : avant d'ajouter une dépendance

Les questions qu'une équipe prudente se pose avant de s'appuyer sur le paquet de quelqu'un d'autre.

- [ ] **En avons-nous besoin ?**
  Quelques lignes de notre propre code peuvent coûter moins cher, à la longue, qu'un paquet qu'il faut suivre et mettre à jour.
- [ ] **Est-il maintenu ?**
  Regardez les versions récentes, les problèmes de sécurité non résolus et le nombre de personnes qui le maintiennent. Un paquet abandonné devient votre problème.
- [ ] **Qu'apporte-t-il avec lui ?**
  Vérifiez combien de dépendances transitives il ajoute ; chacune est du code de plus auquel il faut faire confiance.
- [ ] **La licence est-elle compatible ?**
  Les licences open source diffèrent ; certaines imposent des obligations lorsque vous distribuez votre logiciel.
- [ ] **Les versions sont-elles figées et analysées ?**
  Utilisez un fichier de verrouillage et un outil automatique qui vous avertit lorsqu'une vulnérabilité connue touche une version que vous utilisez.
- [ ] **Pourrions-nous le remplacer ?**
  Placez les dépendances importantes derrière notre propre interface, pour qu'un changement de fournisseur ne touche qu'un seul endroit.

Pourquoi la structure compte : le changement et les équipes

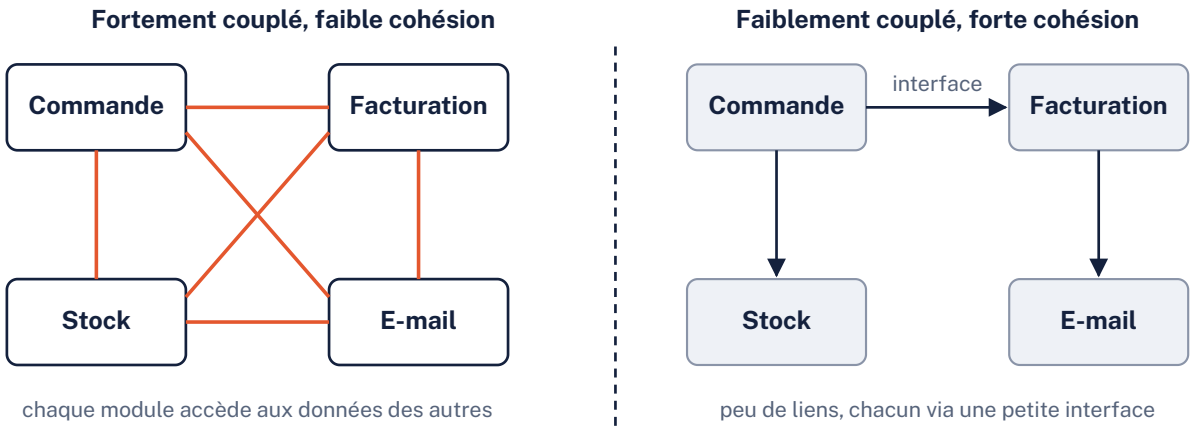
Le code est lu et modifié bien plus souvent qu'il n'est écrit au départ. L'essentiel du coût d'un logiciel sur sa durée de vie part dans sa modification : corriger des bugs, ajouter des fonctionnalités, s'adapter à de nouvelles lois et à de nouveaux partenaires. La structure détermine combien cela coûte. Dans un système bien organisé, une modification du calcul de la TVA touche un seul module. Dans un système mal organisé, elle touche quarante fichiers appartenant à cinq équipes, et quelque chose qui n'a rien à voir casse.

Deux idées, décrites pour la première fois par Larry Constantine et ses collègues dans le mouvement de la conception structurée, à la fin des années 1960 et dans les années 1970, résument ce que signifie « bien organisé ». Le couplage mesure à quel point une partie dépend des détails d'une autre : dans un système fortement couplé, modifier une partie oblige à en

modifier d'autres. La cohésion mesure à quel point le contenu d'une partie va bien ensemble : un module très cohésif fait une seule tâche claire. L'objectif est **un couplage faible et une forte cohésion** : des parties qui font chacune bien une seule tâche et qui communiquent entre elles par de petites interfaces stables.

La structure compte aussi pour les personnes. Si deux équipes doivent toutes deux modifier le même module pour l'essentiel de leur travail, elles passeront leur temps à s'attendre et à se gêner mutuellement. Si chaque équipe possède des modules cohésifs aux interfaces claires, elles peuvent travailler et livrer de façon indépendante. Melvin Conway a observé en 1968 que les systèmes ont tendance à refléter la structure de communication des organisations qui les construisent, une observation connue sous le nom de loi de Conway et étudiée au chapitre 3.

Le couplage en un coup d'œil



Les quatre mêmes modules, reliés de deux façons : chaque lien supplémentaire est un chemin de plus par lequel un changement peut se propager.

ATTENTION

Erreurs courantes

- **Copier-coller au lieu d'écrire une fonction.** La logique dupliquée diverge, si bien que la même règle donne des réponses différentes à différents endroits.
- **Tout est public.** Si chaque détail d'un module peut être utilisé de l'extérieur, chaque détail devient quelque chose dont dépend un autre code, et plus rien ne peut changer sans risque.
- **Du code de framework partout.** Quand les règles métier sont mêlées aux appels au framework, mettre à jour ou remplacer le framework oblige à réécrire le métier.
- **Des dépendances non examinées.** Ajouter un paquet pour une tâche triviale apporte tout son arbre de dépendances, ses bugs et ses risques de sécurité.

Pourquoi c'est important pour l'architecture

L'architecture est en grande partie la même activité que celle de cette section, menée à plus grande échelle et avec des enjeux plus élevés : décider quelles sont les grandes parties d'un système, de quoi chacune est responsable, quelles interfaces les relient et dans quel sens pointent les dépendances. Un module à l'intérieur d'un programme et un service déployé séparément de l'autre côté d'un réseau sont tous deux des parties dotées d'interfaces ; les questions de couplage, de cohésion et de masquage de l'information s'appliquent aux deux.

Le chapitre 3 précise ces principes, en commençant par la modularité ainsi que le couplage et la cohésion, et le chapitre 8 revient sur les dépendances sous l'angle de la sécurité dans la sécurité de la chaîne d'approvisionnement. Ensuite, le code quitte l'ordinateur unique : la section suivante explique comment les ordinateurs communiquent entre eux.

VÉRIFIEZ VOTRE COMPRÉHENSION

▼ **Pourquoi vaut-il mieux appeler une seule fonction `total_with_vat` depuis trois endroits que d'écrire le calcul trois fois ?**

La règle se trouve alors à un seul endroit : quand le taux de TVA change, une seule modification corrige tous les appelants, et les copies ne peuvent pas diverger.

▼ **Qu'est-ce que l'encapsulation empêche le reste du code de faire à un objet compte bancaire ?**

Elle empêche le reste du code de modifier directement le solde, si bien que chaque changement doit passer par des méthodes comme withdraw, qui font respecter les règles.

▼ **Un framework web appelle votre fonction chaque fois qu'une requête pour `/orders` arrive. Comment s'appelle ce fonctionnement ?**

L'inversion de contrôle : c'est le framework, et non votre code, qui contrôle le déroulement et appelle votre code au besoin.

▼ **À quoi sert un fichier de verrouillage, comme `package-lock.json` ?**

Il enregistre la version exacte de chaque paquet installé, y compris les paquets indirects, afin que chaque machine de développeur et chaque build de production utilisent exactement le même code.

▼ **À vous : L'étape de paiement d'une app de livraison de repas appelle directement la bibliothèque d'une société de paiement à douze endroits. Le service financier changera peut-être de société l'an prochain. Comment organiseriez-vous le code pour que ce changement coûte peu, et que coûte votre approche ?**

Définissez votre propre interface de paiement, avec par exemple take_payment et refund, et placez tout le code du prestataire dans un seul module derrière elle. Les douze endroits n'appellent alors que votre interface : un changement revient à écrire une nouvelle implémentation sans toucher au reste. Cela permet aussi aux tests d'utiliser un faux prestataire. Le coût est une couche de plus à écrire et à maintenir, ce qui se justifie ici car le changement est probable et touche à l'argent.

SOURCES

- Python tutorial: modules and packages: https://docs.python.org/3/tutorial/modules.html
- Python tutorial: classes: https://docs.python.org/3/tutorial/classes.html
- Martin Fowler: Inversion of Control: https://martinfowler.com/bliki/InversionOfControl.html
- Semantic Versioning 2.0.0: https://semver.org/
- npm documentation: package-lock.json: https://docs.npmjs.com/cli/v10/configuring-npm/package-lock-json

SECTION 0.4 SUR 0.8 · DÉBUTANT · 25 MIN DE LECTURE · DERNIÈRE RELECTURE : NOVEMBRE 2026

Réseaux, internet et web : IP, DNS, HTTP et le modèle client-serveur

L'IDÉE EN BREF

Envoyer des données sur internet, c'est comme envoyer du courrier. Votre message est découpé en colis (les paquets), chacun portant l'adresse de l'expéditeur et celle du destinataire (les adresses IP) ainsi qu'un numéro de bureau (un port), et des centres de tri (les routeurs) les font suivre. Les noms comme shop.example sont recherchés dans un immense carnet d'adresses partagé (le DNS) pour trouver la véritable adresse. Les lettres elles-mêmes suivent un format fixe (HTTP) : une requête comme « merci de m'envoyer le produit 42 », et une réponse avec un code qui dit comment cela s'est passé. HTTPS met la lettre dans une enveloppe scellée et inviolable.

CE QU'IL FAUT SAVOIR

- Les données traversent les réseaux en paquets, que les routeurs font suivre de saut en saut grâce à l'adresse IP de destination ; un numéro de port identifie le programme sur cette machine.
- TCP fournit une connexion fiable et ordonnée après une poignée de main ; UDP envoie des paquets isolés sans aucune garantie, mais avec moins de délai.
- Le DNS transforme les noms en adresses IP grâce à une hiérarchie de serveurs racine, de domaine de premier niveau et faisant autorité, avec une mise en cache intensive contrôlée par des TTL.
- HTTP est un protocole requête-réponse sans état : une méthode (GET, POST, PUT, DELETE), un chemin, des en-têtes et un corps, auxquels répond un code d'état (2xx, 3xx, 4xx, 5xx).
- HTTPS, c'est HTTP sur TLS, qui chiffre le trafic, détecte les falsifications et utilise des certificats signés par des autorités de confiance pour prouver l'identité du serveur.
- Dans le modèle client-serveur, les clients lancent les requêtes et les serveurs détiennent les données et les règles partagées ; un serveur ne doit jamais faire aveuglément confiance aux clients.
- La latence (le délai) et la bande passante (la capacité) sont deux choses différentes ; la latence est limitée par la vitesse de la lumière, donc la distance et les allers-retours comptent.

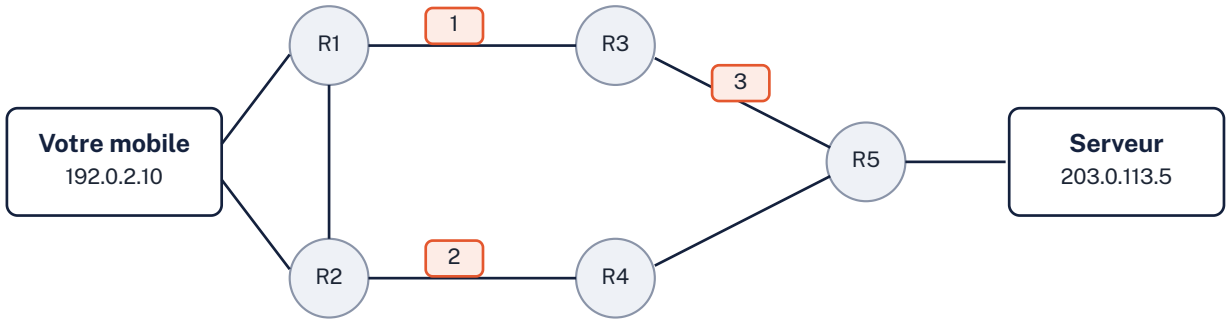
Les réseaux et les paquets

Presque aucun logiciel moderne ne fonctionne seul. Une application de livraison de repas sur votre téléphone dialogue avec les serveurs de l'entreprise, qui dialoguent avec les tablettes des restaurants, un prestataire de paiement, un service de cartographie et un service de notifications. Cette section explique comment fonctionne ce dialogue, en partant du bas.

Un **réseau** est un ensemble d'ordinateurs reliés de façon à pouvoir échanger des données, par câble, Wi-Fi, réseau mobile ou fibre optique. **Internet** est un réseau de réseaux : des millions de réseaux appartenant à des propriétaires différents (box internet à domicile, opérateurs mobiles, universités, fournisseurs de cloud) qui ont accepté de transporter le trafic les uns des autres selon des règles communes appelées *protocoles*.

Les données ne circulent pas en un seul flux continu. Elles sont découpées en paquets : des morceaux d'environ 1500 octets au maximum en général, chacun étiqueté avec sa provenance et sa destination, comme un colis portant l'adresse de l'expéditeur et celle du destinataire. Des appareils appelés **routeurs** font passer les paquets d'un réseau à l'autre, chacun choisissant un bon saut suivant, un peu comme des centres de tri qui font suivre les colis vers leur destination. Différents paquets d'un même message peuvent emprunter des routes différentes, arriver dans le désordre ou parfois se perdre, et l'ordinateur destinataire les remet dans l'ordre. Découper les données en paquets permet à des millions de conversations de partager équitablement les mêmes câbles, et permet au trafic de contourner une liaison coupée.

Les paquets trouvent leur propre chemin



Un message découpé en paquets numérotés ; chacun passe par ses routeurs, puis tout est réassemblé

Aucun fil unique ne relie votre téléphone au serveur ; les paquets sautent entre des routeurs appartenant à de nombreux réseaux différents.

Les adresses : adresses IP et ports

Pour qu'un paquet arrive, la destination a besoin d'une adresse. Chaque appareil sur internet est joignable par une adresse IP (IP signifie Internet Protocol). L'ancienne version, IPv4, écrit les adresses sous la forme de quatre nombres de 0 à 255, comme `192.0.2.10` ; elle permet environ 4,3 milliards d'adresses, ce qui ne suffit pas pour tous les téléphones, ordinateurs portables et capteurs du monde. La nouvelle version, IPv6, utilise des adresses beaucoup plus longues écrites en hexadécimal (base 16, avec les chiffres 0 à 9 et les lettres a à f), comme `2001:db8::1`, et peut accueillir un nombre pratiquement illimité d'appareils. Les deux sont utilisées côte à côte. (Les adresses d'exemple de cette section proviennent de plages réservées à la documentation : elles n'appartiennent à aucune machine réelle.)

Une adresse IP amène un paquet au bon ordinateur, mais un ordinateur exécute de nombreux programmes à la fois. Un numéro de port, de 0 à 65 535, indique quel programme doit le recevoir. Si l'adresse IP est l'adresse postale d'un immeuble de bureaux, le port est le numéro du bureau. Les services courants ont des ports standard : 443 pour HTTPS, 80 pour HTTP simple, 53 pour le DNS, 5432 pour la base de données PostgreSQL. Un programme serveur *écoute* sur un port, en attendant des connexions ; quand votre navigateur contacte un site web, il se connecte au port 443 de l'adresse IP du site.

TCP et UDP : deux façons de livrer

Au-dessus d'IP se trouvent deux grands protocoles de transport, qui décident du soin apporté à la livraison. TCP (Transmission Control Protocol) offre à deux programmes un flux d'octets fiable et ordonné. Avant tout envoi de données, les deux côtés établissent une connexion par une *poignée de main* (handshake) en trois étapes. Chaque paquet est numéroté ; le destinataire accuse réception de ce qui est arrivé ; tout ce qui manque est renvoyé ; tout est remis au programme dans le bon ordre. TCP ralentit aussi quand le réseau est encombré. C'est comme un envoi recommandé avec suivi : fiable, mais avec un certain surcoût.

UDP (User Datagram Protocol) envoie simplement des paquets isolés, appelés datagrammes, sans poignée de main, sans accusé de réception, sans renvoi et sans ordre. C'est comme envoyer des cartes postales : bon marché et rapide, mais certaines peuvent ne pas arriver. C'est exactement ce qu'il faut quand un paquet en retard ne sert de toute façon plus à rien. Dans un appel vidéo, un fragment de son qui arrive avec une demi-seconde de retard est pire qu'un fragment qui n'arrive jamais : les appels, les diffusions en direct et les jeux en ligne utilisent donc UDP et s'accommodent de petites lacunes. Les requêtes DNS, qui sont de toutes petites questions et réponses, utilisent elles aussi traditionnellement UDP.

TCP et UDP comparés

	TCP	UDP
Établissement de la connexion	Oui : une poignée de main avant l'envoi des données	Aucun : on envoie, c'est tout
Livraison	Garantie : les paquets perdus sont renvoyés	Au mieux (best effort) : les pertes ne sont pas réparées
Ordre	Préservé	Non garanti
Surcoût et délai	Plus élevés	Plus faibles
Usages typiques	Pages web et API (HTTP/1.1 et HTTP/2), e-mail, transfert de fichiers, connexions aux bases de données	Appels vocaux et vidéo, diffusion en direct, jeux en ligne, DNS

POUR ALLER PLUS LOIN

Pour aller plus loin : QUIC et HTTP/3

La version la plus récente du protocole du web, HTTP/3 (publiée sous le nom de RFC 9114 en 2022), ne fonctionne pas sur TCP. Elle fonctionne sur QUIC (RFC 9000, 2021), un protocole de transport construit au-dessus d'UDP, qui réimplémente lui-même la livraison fiable et combine l'établissement de la connexion avec celui du chiffrement, ce qui économise des allers-retours. Cela montre que la frontière entre « TCP fiable » et « UDP non fiable » porte sur ce que garantit la couche transport, et que la fiabilité peut être construite à une couche supérieure lorsque cela sert mieux l'application.

Le DNS : le carnet d'adresses d'internet

Les personnes retiennent des noms comme `shop.example` ; les réseaux acheminent selon des adresses IP. Le Domain Name System (DNS, système de noms de domaine) traduit les uns en les autres. C'est un annuaire mondial et distribué : aucun ordinateur ne détient à lui seul tous les noms. La responsabilité est au contraire répartie selon les points du nom, lus de droite à gauche. Les serveurs **racine** savent qui est responsable de chaque domaine de premier niveau, comme `.com` ou `.uk` ; ces serveurs de domaine de premier niveau savent quels serveurs de noms **faisant autorité** sont responsables de chaque domaine enregistré sous eux ; et les serveurs faisant autorité détiennent les enregistrements eux-mêmes, comme l'adresse IP de `www.shop.example`. La zone racine est servie par 13 identités de serveurs nommées, exploitées par 12 organisations indépendantes au moyen de bien plus d'un millier d'instances de serveurs dans le monde.

En général, votre appareil ne parcourt pas lui-même cette chaîne. Il interroge un **résolveur récursif**, généralement exploité par votre fournisseur d'accès à internet ou par un service public, qui fait le travail et mémorise (met en cache) les réponses pendant une durée fixée par le propriétaire du domaine, appelée TTL (time to live, durée de vie). Comme les réponses sont mises en cache à de nombreux niveaux, la plupart des recherches obtiennent une réponse en quelques millisecondes sans jamais atteindre la racine. Cette même mise en cache explique pourquoi la modification d'un enregistrement DNS peut mettre des minutes, voire des heures, à être visible partout.

La résolution DNS étape par étape (rien en cache)

1. **Interroger le résolveur**
   Votre appareil demande au résolveur récursif configuré : quelle est l'adresse IP de `www.shop.example` ?

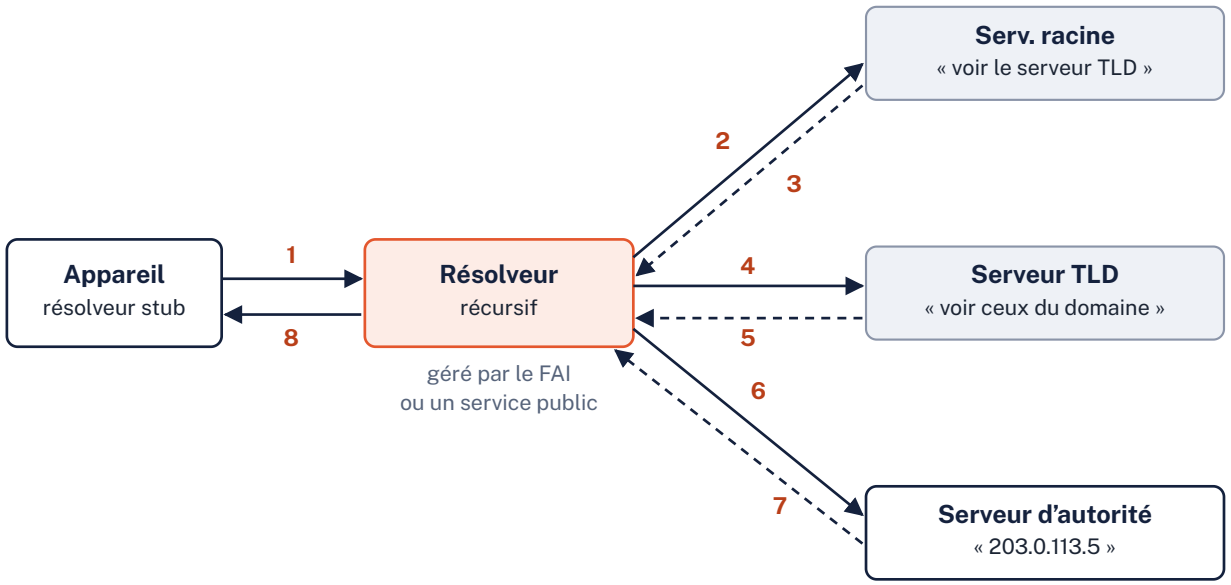
2. **Interroger un serveur racine**
   Le résolveur interroge un serveur racine, qui ne connaît pas la réponse, mais lui indique les serveurs responsables du domaine de premier niveau.

3. **Interroger le serveur du domaine de premier niveau**
   Le résolveur interroge l'un de ces serveurs, qui lui indique les serveurs de noms faisant autorité pour `shop.example`.

4. **Interroger le serveur faisant autorité**
   Le résolveur interroge le serveur faisant autorité, qui répond avec l'enregistrement : par exemple `203.0.113.5`, valable 300 secondes.

5. **Répondre et mémoriser**
   Le résolveur renvoie l'adresse à votre appareil, et tous deux la gardent en cache jusqu'à expiration du TTL, si bien que la recherche suivante est quasi instantanée.

Qui répond à une question DNS



Les flèches pleines sont des questions, les flèches en pointillés des réponses ; grâce au cache, la plupart des recherches réelles sautent les étapes 2 à 7.

HTTP : requêtes et réponses

Une fois qu'un programme peut joindre un autre programme, les deux ont besoin d'un langage commun pour dire ce qu'ils veulent. Sur le web, ce langage est HTTP (Hypertext Transfer Protocol). Il suit un schéma simple : le client envoie une **requête** ; le serveur renvoie une **réponse**. Chaque requête indique une **méthode** (quoi faire), un **chemin** (sur quoi), des **en-têtes** (des informations supplémentaires sous forme de paires nom-valeur) et parfois un **corps** (les données envoyées). Chaque réponse porte un code d'état, des en-têtes et généralement un corps : une page web, une image ou des données dans un format comme JSON (un format texte simple pour les données structurées, expliqué dans la section sur les API).

Les méthodes les plus courantes sont `GET` (récupérer quelque chose, sans le modifier), `POST` (soumettre quelque chose de nouveau, comme passer une commande), `PUT` (remplacer quelque chose à une adresse connue, comme mettre à jour une adresse de livraison) et `DELETE` (supprimer quelque chose). GET, PUT et DELETE sont par définition *idempotentes* : envoyer deux fois la même requête a le même effet que l'envoyer une fois, ce qui permet de les retenter sans risque après une panne réseau. POST ne l'est pas, et c'est pourquoi une nouvelle tentative imprudente peut passer une commande deux fois. Les en-têtes transportent des éléments comme le type de contenu (`Content-Type: application/json`), la langue préférée de l'utilisateur, les cookies (de petits éléments de données qu'un site demande au navigateur de stocker et de renvoyer avec les requêtes suivantes) et les identifiants, ainsi que des instructions de mise en cache.

HTTP est *sans état* : chaque requête se comprend par elle-même, et le serveur n'est pas tenu de se souvenir des précédentes. Les sessions de connexion sont construites par-dessus, généralement par l'envoi d'un cookie ou d'un jeton par le client à chaque requête. L'absence d'état est l'une des raisons pour lesquelles le web passe si bien à l'échelle : n'importe lequel de nombreux serveurs identiques peut traiter n'importe quelle requête.

```
POST /api/orders HTTP/1.1
Host: shop.example
Content-Type: application/json
Authorization: Bearer eyJhbGciOi...

{"product_id": 42, "quantity": 2}

HTTP/1.1 201 Created
Content-Type: application/json
Location: /api/orders/9917

{"order_id": 9917, "status": "received", "total_pence": 3000}
```

Un échange HTTP réaliste : passer une commande, puis la réponse du serveur

Les codes d'état par famille

Famille	Signification	Exemples courants
2xx	Succès	200 OK ; 201 Created (quelque chose de nouveau a été créé) ; 204 No Content
3xx	Redirection : cherchez ailleurs	301 Moved Permanently ; 302 Found (déplacement temporaire) ; 304 Not Modified (votre copie en cache est toujours valable)
4xx	Erreur du client : la requête était incorrecte	400 Bad Request ; 401 Unauthorized (non connecté) ; 403 Forbidden (connecté mais non autorisé) ; 404 Not Found ; 429 Too Many Requests
5xx	Erreur du serveur : la requête était correcte, le serveur a échoué	500 Internal Server Error ; 502 Bad Gateway ; 503 Service Unavailable ; 504 Gateway Timeout

La distinction entre 4xx et 5xx compte en pratique. Un 4xx signifie généralement « ne vous contentez pas de réessayer ; corrigez la requête » ; les exceptions sont 408 (la requête a expiré) et 429 (trop de requêtes), pour lesquels il vaut la peine de réessayer après un délai. Un 5xx signifie « le problème vient de chez nous » ; réessayer plus tard peut fonctionner, et l'équipe d'exploitation doit être alertée quand le taux de 5xx augmente.

HTTPS et TLS : confidentialité et identité

HTTP simple envoie tout sous forme de texte lisible, si bien que n'importe qui sur le trajet, par exemple une personne connectée au même Wi-Fi d'un café, pourrait le lire ou le modifier. HTTPS, c'est HTTP envoyé dans un canal protégé créé par **TLS** (Transport Layer Security ; la version actuelle, TLS 1.3, a été publiée sous le nom de RFC 8446 en 2018). TLS apporte trois choses : la *confidentialité* (les personnes qui écoutent ne voient que des données brouillées), l'*intégrité* (toute falsification est détectée) et l'*authentification* (vous parlez vraiment à `shop.example`, et non à un imposteur).

L'authentification repose sur des **certificats**. Le certificat d'un site est un document numérique qui dit en substance : « cette clé publique appartient à `shop.example` », signé par une **autorité de certification** (CA) à laquelle votre navigateur ou votre téléphone fait déjà confiance. Il fonctionne comme un passeport : délivré par une autorité à laquelle les deux parties font confiance, vérifié par rapport à son titulaire, et valable seulement jusqu'à sa date d'expiration. Pendant la poignée de main TLS, le serveur présente son certificat, le client vérifie la signature, le nom et les dates, et les deux parties conviennent de nouvelles clés secrètes pour chiffrer le reste de la conversation. Si une vérification échoue, le navigateur affiche un avertissement.

TLS protège les données *en transit* entre deux points. Il ne protège pas les données une fois stockées, et il ne prouve pas que le site est honnête, seulement qu'il est bien celui qu'il prétend être. Aujourd'hui, HTTPS est la norme pour tout le trafic web, pas seulement pour les paiements, et les certificats sont souvent délivrés gratuitement et renouvelés automatiquement.

Le modèle client-serveur

La plupart des systèmes de ce cours suivent le modèle client-serveur. Un **client** (un navigateur, une application mobile, un autre programme) entame la conversation en envoyant une requête. Un **serveur** attend des requêtes et y répond. Le serveur détient ce qui doit être partagé et digne de confiance : le catalogue de produits, les commandes de tout le monde, les règles sur les prix et les remboursements. Le client gère l'écran de l'utilisateur et ses saisies.

Dans ce même restaurant, vu depuis la salle, les convives sont les clients et la cuisine dans son ensemble est le serveur : les convives passent commande ; la cuisine détient les ingrédients et le savoir-faire et sert de nombreux convives à la fois. Une conséquence est que le serveur ne doit jamais faire aveuglément confiance au client. Une application mobile peut être modifiée par son utilisateur : les prix, les autorisations et les niveaux de stock doivent donc être vérifiés sur le serveur, même si l'application les vérifie elle aussi avec soin.

Un serveur est souvent aussi un client : le serveur web de la boutique est client de sa base de données et du prestataire de paiement. Les systèmes réels sont des chaînes et des toiles de telles relations, comme le montre le diagramme de séquence ci-dessous.

Latence et bande passante

Deux chiffres décrivent la rapidité ressentie d'un réseau, et on les confond souvent. La bande passante est la quantité de données qu'une connexion peut transporter par seconde, par exemple 100 mégabits par seconde : la largeur du tuyau. La latence est le temps que met une donnée pour aller d'un bout à l'autre, généralement mesuré comme un **temps d'aller-retour** (RTT, round-trip time) en millisecondes : la longueur du tuyau. Une analogie autoroutière aide : ajouter des voies (la bande passante) laisse passer plus de voitures par heure, mais ne raccourcit le trajet d'aucune voiture (la latence).

La latence a un plancher fixé par la physique. Dans la fibre optique, la lumière se déplace à environ 200 000 km par seconde, soit environ les deux tiers de sa vitesse dans le vide. Londres et Sydney sont distantes d'environ 17 000 km en ligne droite à la surface du globe, si bien que même une fibre parfaite suivant cette ligne demanderait environ 85 ms dans un sens et 170 ms pour un aller-retour. Les câbles réels ne suivent pas le plus court chemin et les routeurs ajoutent du délai : les allers-retours mesurés sont donc généralement d'environ 250 ms. Aucune mise à niveau ne peut supprimer cela. Si le chargement d'une page depuis Sydney exige dix allers-retours successifs vers un serveur à Londres, l'utilisateur attend deux secondes et demie, quelle que soit la vitesse de sa connexion internet.

EXEMPLE COMMENTÉ

Exemple commenté : un serveur à Londres, un client à Sydney

Une boutique héberge tout à Londres. Un client à Sydney ouvre une page produit via une nouvelle connexion HTTPS. On suppose un aller-retour de 250 ms et on ignore le temps de traitement du serveur.

Recherche DNS (pas en cache)	Le résolveur peut avoir besoin de plusieurs requêtes ; disons 1 aller-retour vers des serveurs proches : environ 50 ms
Poignée de main TCP	1 aller-retour vers Londres : 250 ms
Poignée de main TLS 1.3	1 aller-retour : 250 ms
Demander la page HTML	1 aller-retour : 250 ms
Demander le CSS et les scripts dont la page a besoin	Au moins 1 aller-retour de plus : 250 ms
Total avant que quoi que ce soit d'utile n'apparaisse	Environ 1050 ms, la bande passante ne jouant presque aucun rôle
Même client, contenu servi depuis Sydney	Aller-retour d'environ 10 ms : les cinq mêmes étapes prennent à peu près 50 ms

Pour les petites pages, c'est la latence, et non la bande passante, qui domine. Faire passer la connexion du client de 50 à 500 mégabits par seconde ne changerait presque rien au résultat.

Les architectes réduisent la latence en supprimant des allers-retours (en réutilisant les connexions, en regroupant les requêtes) et en rapprochant le contenu des utilisateurs, par exemple avec des réseaux de diffusion de contenu, traités dans Les serveurs et le cloud.

GUIDE DE DÉCISION

Guide de décision : une expérience réseau lente, est-ce une question de latence ou de bande passante ?

Un premier diagnostic approximatif quand les utilisateurs se plaignent d'une lenteur sur le réseau.

Q1 **Les petites requêtes (un court appel d'API, une page minuscule) sont-elles lentes elles aussi ?**

- **Oui** La latence est la cause probable. Vérifiez la distance jusqu'au serveur et le nombre d'allers-retours successifs effectués par la page ou l'application.
- **Non** Passez à la question suivante.

Q2 **Seuls les gros transferts (vidéos, gros téléchargements, rapports volumineux) sont-ils lents ?**

- **Oui** La bande passante est la limite probable, sur la connexion de l'utilisateur ou sur celle du serveur. Envisagez la compression, des fichiers plus petits ou un réseau de diffusion de contenu.
- **Non** Passez à la question suivante.

Q3 **Le délai est-il le même depuis tous les endroits ?**

- **Oui** Le réseau n'est peut-être pas du tout en cause : examinez le traitement côté serveur, les requêtes à la base de données et les algorithmes (voir la section sur les données et les algorithmes).
- **Non** Un délai qui dépend du lieu indique un problème de latence et de routage : servez les utilisateurs depuis un endroit plus proche d'eux.

Ce qui se passe quand vous tapez une adresse web

Tous ces éléments se combinent chaque fois que vous visitez un site web. Voici l'enchaînement lorsque vous tapez `https://shop.example/products/42` et appuyez sur Entrée, de façon simplifiée mais exacte.

De la touche Entrée à la page

1. **Analyser l'adresse**
   Le navigateur découpe l'URL en schéma (`https` , donc port 443 et TLS), nom d'hôte (`shop.example`) et chemin (`/products/42`).

2. **Trouver l'adresse IP**
   Le navigateur vérifie son propre cache, puis celui du système d'exploitation, puis interroge le résolveur récursif, qui consulte le DNS comme décrit plus haut et renvoie, par exemple, `203.0.113.5`.

3. **Ouvrir une connexion TCP**
   Le navigateur et le serveur échangent les trois messages de la poignée de main pour établir une connexion fiable vers le port 443.

4. **Établir TLS**
   Le serveur présente son certificat ; le navigateur vérifie qu'il est valable pour `shop.example` et signé par une autorité de confiance ; les deux parties conviennent de clés de chiffrement.

5. **Envoyer la requête HTTP**
   Le navigateur envoie `GET /products/42` avec des en-têtes comme les langues acceptées et les éventuels cookies du site.

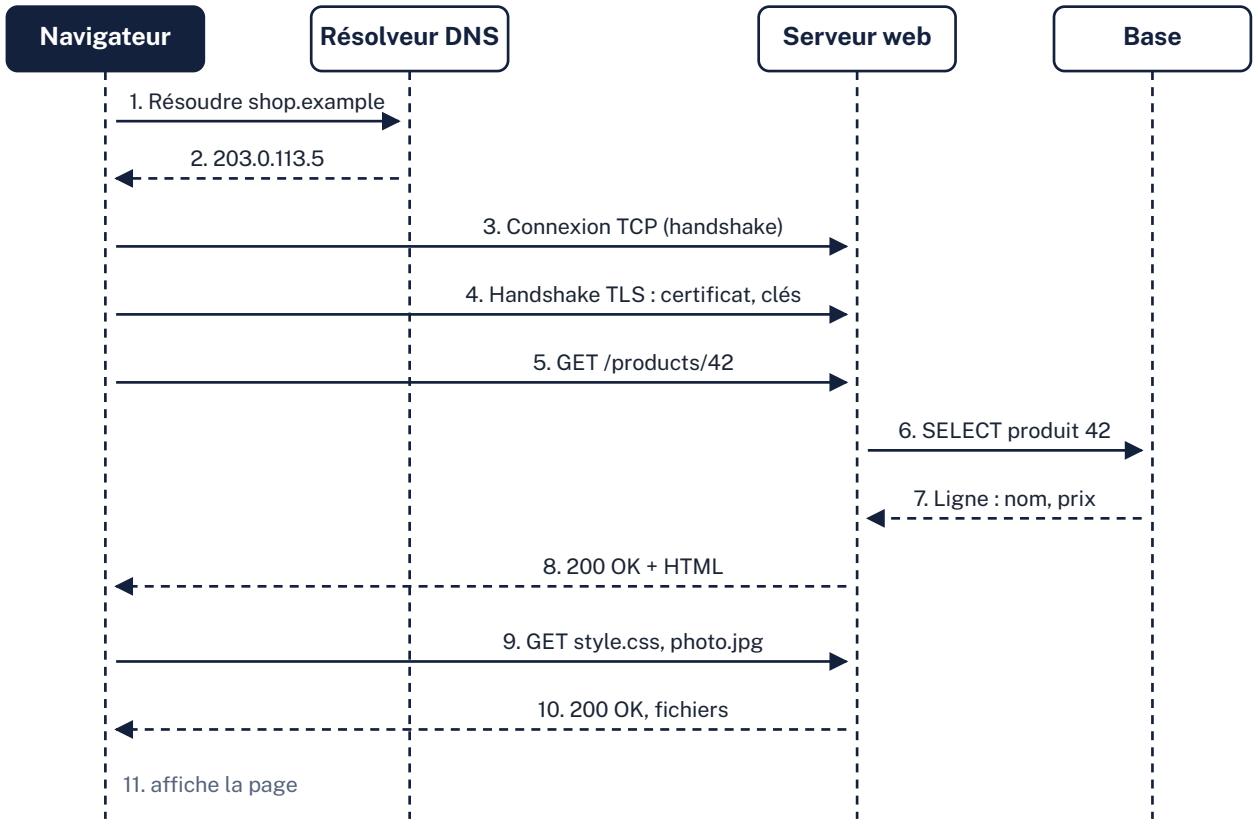
6. **Le serveur fait son travail**
   Le logiciel serveur reçoit la requête et exécute le code de la boutique, qui interroge une base de données pour obtenir le produit 42, puis construit une page HTML.

7. **Recevoir la réponse**
   Le serveur répond `200 OK` avec le HTML dans le corps et des en-têtes comme `Content-Type: text/html` et des règles de mise en cache.

8. **Récupérer le reste**
   Le HTML fait référence à des feuilles de style, des scripts et des images ; le navigateur les demande aussi, souvent plusieurs à la fois sur la même connexion.

9. **Afficher**
   Le navigateur met en page et dessine la page à l'écran ; des scripts peuvent ensuite faire d'autres requêtes, par exemple pour récupérer des avis.

Diagramme de séquence : le chargement d'une page produit



Le temps s'écoule vers le bas ; les flèches pleines sont des requêtes, les flèches en pointillés des réponses.

ATTENTION

Idées reçues courantes

- « **Internet est rapide, donc la distance n'a pas d'importance.** » La bande passante a énormément augmenté ; la vitesse de la lumière, non. Chaque aller-retour vers un serveur lointain coûte un temps bien réel.
- « **HTTPS signifie que le site est sûr.** » Cela signifie que la connexion est privée et que le site est bien celui qu'il prétend être. Un site frauduleux peut avoir un certificat valide.
- « **Un 404 signifie que le serveur est en panne.** » Un 404, c'est le serveur qui fonctionne correctement et qui dit que la chose demandée n'existe pas. Un serveur en panne ne donne généralement aucune réponse, ou bien un 5xx venant d'un élément placé devant lui.
- « **Modifier le DNS est instantané.** » Les réponses en cache restent valables jusqu'à l'expiration de leur TTL : les modifications se propagent donc progressivement.

Pourquoi c'est important pour l'architecture

Dès qu'un système est réparti sur plus d'un ordinateur, chaque interaction entre ses parties devient un appel réseau, avec de la latence, un risque de défaillance et un besoin de sécurité. Un appel de fonction à l'intérieur d'un programme prend des nanosecondes et ne se perd jamais ; un appel à travers un réseau prend des millisecondes, parfois bien plus, et parfois ne revient jamais. Beaucoup des problèmes les plus difficiles de l'architecture découlent de cette différence, et c'est pourquoi le chapitre 6 s'ouvre sur les illusions de l'informatique distribuée, comme la croyance que la latence est nulle.

Les protocoles vus ici reviennent aussi comme briques de base : HTTP sous-tend les API web de la partie suivante de ce chapitre et les styles de communication du chapitre 6, tandis que TLS et les certificats reviennent dans la communication sécurisée et le zero trust. Placer des serveurs près des utilisateurs relève de l'architecture multirégion et en périphérie (edge).

VÉRIFIEZ VOTRE COMPRÉHENSION

▼ **Pourquoi un paquet a-t-il besoin d'un numéro de port en plus d'une adresse IP ?**

L'adresse IP l'amène au bon ordinateur, mais cet ordinateur exécute de nombreux programmes ; le port indique quel programme doit le recevoir, par exemple le serveur web qui écoute sur le port 443.

▼ **Que dit au client un code d'état 3xx ?**

Que ce qu'il a demandé se trouve ailleurs ou n'a pas changé : par exemple, 301 signifie que la ressource a été déplacée définitivement, et 304 que la copie en cache du client est toujours valable.

▼ **Quelle est la différence entre une réponse 401 et une réponse 503 ?**

401 est une erreur du client : la requête ne comporte pas d'identifiants valides. 503 est une erreur du serveur : le service est temporairement incapable de traiter la requête.

▼ **Pourquoi aucune mise à niveau du réseau ne peut-elle ramener un aller-retour Londres-Sydney à 10 ms ?**

Dans la fibre, la lumière parcourt environ 200 000 km par seconde : l'aller-retour d'environ 34 000 km prend donc au moins 170 ms environ.

▼ **À vous : La page d'une boutique hébergée à Londres nécessite quatre allers-retours sur une nouvelle connexion HTTPS : TCP, TLS, HTML et CSS. Une cliente à Sydney a un aller-retour de 240 ms vers Londres, ou de 20 ms vers un CDN local. Comparez les attentes et donnez votre conseil.**

Depuis Londres : 4 x 240 ms = 960 ms avant que quoi que ce soit d'utile n'apparaisse. Depuis le CDN : 4 x 20 ms = 80 ms, sans compter le DNS et le temps du serveur. Une connexion plus rapide n'aiderait pas, car le délai vient de la distance et des allers-retours, non de la bande passante. Conseillez de servir le HTML statique, le CSS et les images depuis le CDN, et rappelez-vous que tout ce qui est personnel ou change souvent doit encore remonter jusqu'à l'origine.

SOURCES

- MDN: An overview of HTTP: https://developer.mozilla.org/en-US/docs/Web/HTTP/Guides/Overview
- MDN: HTTP response status codes: https://developer.mozilla.org/en-US/docs/Web/HTTP/Reference/Status
- RFC 9110: HTTP Semantics: https://www.rfc-editor.org/rfc/rfc9110
- RFC 8446: The Transport Layer Security (TLS) Protocol Version 1.3: https://www.rfc-editor.org/rfc/rfc8446
- Root Server Technical Operations Association: https://root-servers.org/

SECTION 0.5 SUR 0.8 · DÉBUTANT · 24 MIN DE LECTURE · DERNIÈRE RELECTURE : NOVEMBRE 2026

Fichiers et bases de données : stockage relationnel et non relationnel, requêtes et transactions

L'IDÉE EN BREF

Une base de données ressemble à une bibliothèque tenue par un bibliothécaire soigneux. Au lieu que chaque programme range ses papiers dans son propre tiroir, toutes les données vont à un seul endroit qui tient un catalogue (un index), vérifie que chaque enregistrement respecte les règles, veille à ce que deux personnes ne puissent pas modifier la même chose en même temps, et ne perd jamais une modification qu'il a promis de conserver. La plupart des bases de données stockent les données dans des tables, comme des tableurs aux règles strictes, et on les interroge dans un langage appelé SQL.

CE QU'IL FAUT SAVOIR

- Les fichiers conviennent aux documents entiers, comme les images et les journaux ; les bases de données conviennent aux données structurées que de nombreux utilisateurs lisent et modifient en même temps, car le SGBD gère la concurrence, les règles, la recherche rapide et la reprise après une panne.
- Les bases de données relationnelles stockent les données dans des tables faites de lignes et de colonnes ; une clé primaire identifie chaque ligne et une clé étrangère relie une ligne à une ligne d'une autre table.
- SQL est déclaratif : vous indiquez le résultat voulu, par exemple un SELECT filtré ou un JOIN entre plusieurs tables, et le planificateur de requêtes de la base de données décide comment l'obtenir.
- Un index permet à la base de données de trouver des lignes en O(log n) étapes au lieu de parcourir toute la table, au prix d'un stockage supplémentaire et d'écritures plus lentes.
- Une transaction fait réussir ou échouer ensemble plusieurs modifications ; ACID (atomicité, cohérence, isolation, durabilité) décrit ses garanties, comme dans un virement bancaire qui ne peut jamais se faire à moitié.
- Les familles NoSQL (documents, clé-valeur, colonnes larges et graphes) conviennent à des formes de données et à des modes d'accès particuliers ; de nombreux systèmes utilisent une base relationnelle plus une ou plusieurs de ces bases.
- Une sauvegarde n'est prouvée que lorsqu'on la restaure, les répliques ne protègent pas contre les erreurs, et les bases de données managées dans le cloud prennent en charge l'exploitation, mais pas la conception du schéma et des requêtes.

Pourquoi les programmes ont besoin d'un endroit où garder les données

Un programme en cours d'exécution garde ses données de travail en mémoire (RAM), qui est rapide mais oublie tout quand le programme s'arrête ou que la machine redémarre (voir ce qu'est un logiciel). Tout ce qui doit survivre à un redémarrage, comme le compte d'un client, une commande ou un rendez-vous à l'hôpital, doit être écrit sur un support qui conserve son contenu sans alimentation : un disque dur ou un SSD. Les données qui survivent ainsi sont dites *persistantes*, et décider comment les rendre persistantes est l'une des premières décisions de tout système, et l'une de celles qui durent le plus longtemps.

La façon la plus simple de rendre des données persistantes est un fichier : une suite d'octets portant un nom, que le système d'exploitation stocke sur disque. Les fichiers sont excellents pour ce qui forme naturellement un document entier : images, PDF, vidéos, configuration, journaux, rapports exportés. Les systèmes les conservent souvent dans un *stockage objet* dans le cloud (un service qui stocke des fichiers par leur nom) et n'enregistrent ailleurs qu'une référence.

Les fichiers deviennent pénibles quand de nombreux utilisateurs lisent et modifient en même temps de petits morceaux de données structurées. Imaginez une boutique qui garde toutes ses commandes dans un seul fichier de tableur. Deux clients passent au paiement au même moment : quelle modification l'emporte ? Le serveur tombe en panne au milieu d'un enregistrement : le fichier est-il à moitié ancien, à moitié nouveau ? Une personne du service client veut les commandes d'un seul client : le programme doit-il lire tout le fichier ? Quelqu'un saisit un numéro de client qui n'existe pas : qui l'en empêche ? Chaque programme utilisant le fichier devrait résoudre chacun de ces problèmes, et de la même façon.

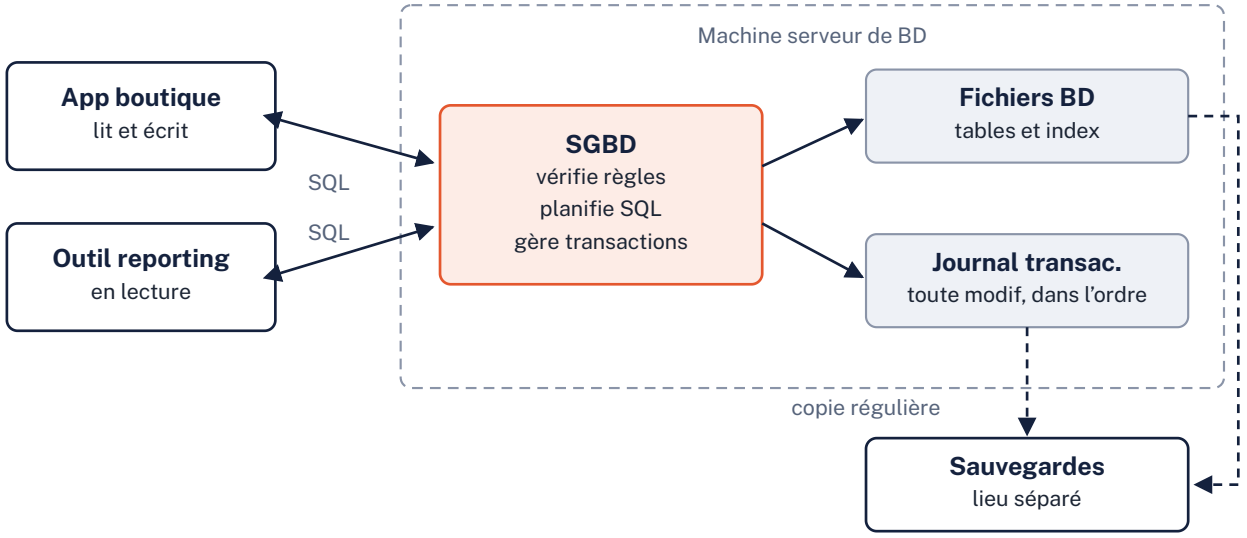
Ce qu'est une base de données

Une base de données est un ensemble organisé de données, et le programme qui s'en occupe est un système de gestion de base de données (SGBD), comme PostgreSQL, MySQL, Microsoft SQL Server, Oracle Database ou SQLite. On dit souvent « la base de données » pour les deux. Le SGBD résout les problèmes des fichiers une fois pour toutes, pour tout le monde : il permet à de nombreux utilisateurs de travailler en même temps sans se marcher sur les pieds, trouve les données rapidement, refuse les données qui enfreignent les règles et revient à un état correct après une panne.

L'analogie est celle d'une bibliothèque. Une pile de papiers sur un bureau suffit si vous êtes seul à l'utiliser. Une bibliothèque a un catalogue, des règles sur l'emplacement de chaque chose, un registre de qui a emprunté quoi, et un bibliothécaire qui ne laissera pas deux personnes prendre le même exemplaire. Les applications ne touchent pas aux rayonnages : elles s'adressent au bibliothécaire.

Dans la plupart des systèmes d'entreprise, le SGBD tourne comme un processus serveur à part, souvent sur sa propre machine, et les applications lui parlent par le réseau (voir comment les ordinateurs communiquent). SQLite est une exception bien connue : c'est une bibliothèque qui s'exécute à l'intérieur de l'application et stocke toute la base de données dans un seul fichier, ce qui explique qu'on la trouve dans les téléphones, les navigateurs et les petits outils.

Où les données se trouvent réellement



Les applications n'écrivent jamais directement dans les fichiers de la base de données : elles envoient des requêtes au SGBD, qui écrit les fichiers de données et un journal, et un processus distinct copie les deux vers un stockage de sauvegarde.

Bases de données relationnelles : tables, clés et relations

Le type de base de données le plus utilisé est la base de données relationnelle, fondée sur un modèle publié par Edgar F. Codd, d'IBM, en 1970. Les données sont rangées dans des **tables**. Chaque table décrit un type de chose (clients, commandes, produits). Chaque **ligne** est un exemplaire de cette chose (un client), et chaque **colonne** est un fait à son sujet (nom, adresse e-mail, date d'inscription), avec un type déclaré comme entier, texte, date ou décimal.

Chaque table a une clé primaire : une colonne (ou une combinaison de colonnes) dont la valeur est unique pour chaque ligne et jamais vide, comme `customer_id`. Une clé étrangère est une colonne d'une table qui contient la clé primaire d'une ligne d'une autre table. Une commande a une colonne `customer_id`, qui indique quel client l'a passée. La déclarer comme clé étrangère indique au SGBD de refuser une commande pour le client 999 s'il n'existe pas de client 999 : c'est ce qu'on appelle l'*intégrité référentielle*.

L'ensemble des tables, colonnes, types, clés et règles forme le schéma. Le schéma est un contrat : chaque programme qui utilise la base de données peut compter sur le fait qu'une commande a toujours un client valide et un prix qui est un nombre. Modifier un schéma plus tard, quand des millions de lignes et plusieurs programmes en dépendent, est possible mais demande du soin ; c'est pourquoi l'évolution des schémas a sa propre section au chapitre 7.

Les conceptions relationnelles cherchent à stocker chaque fait une seule fois. L'adresse e-mail du client se trouve dans une ligne de la table des clients, et n'est pas recopiée dans chaque commande. C'est ce qu'on appelle la *normalisation*. Ainsi, une mise à jour se fait à un seul endroit et ne peut pas laisser derrière elle des copies contradictoires. Le prix de cet ordre est que répondre à une question oblige souvent à combiner des tables, et c'est à cela que servent les jointures.

Poser des questions avec SQL

SQL (Structured Query Language, prononcé « S-Q-L » ou, à l'anglaise, « sequel ») est le langage utilisé pour travailler avec les bases de données relationnelles. Il a été développé chez IBM dans les années 1970 et est devenu une norme ANSI et ISO à la fin des années 1980 ; tous les grands SGBD relationnels le parlent, chacun avec de petites différences de dialecte. SQL est *déclaratif* : vous décrivez le résultat voulu, pas les étapes pour le trouver. Le *planificateur de requêtes* du SGBD choisit les étapes, par exemple lire toute une table ou utiliser un index.

Une jointure (join) combine des lignes de deux tables ou plus en faisant correspondre des colonnes, en général une clé étrangère avec la clé primaire à laquelle elle renvoie. La seconde requête ci-dessous liste chaque ligne des commandes d'un client avec le nom du produit, en puisant dans trois tables à la fois.

```
-- One customer's orders since 1 October, newest first
SELECT order_id, status, placed_at
FROM orders
WHERE customer_id = 42
  AND placed_at ⩾ '2026-10-01'
ORDER BY placed_at DESC;

-- Each order line with the product's name (a join)
SELECT o.order_id, p.name, i.quantity, i.unit_price
FROM orders AS o
JOIN order_items AS i ON i.order_id = o.order_id
JOIN products AS p ON p.product_id = i.product_id
WHERE o.customer_id = 42;
```

Deux requêtes sur le schéma de la boutique présenté plus loin dans cette section : un SELECT filtré et un JOIN sur trois tables.

`INSERT`, `UPDATE` et `DELETE` modifient les données, et `CREATE TABLE` définit le schéma. Le code des applications envoie souvent le SQL par l'intermédiaire d'un *outil de mapping objet-relationnel* (ORM), une bibliothèque qui transforme les lignes en objets. Les ORM évitent d'écrire beaucoup de code, mais cachent le SQL réellement envoyé ; un problème de performances classique est un ORM qui envoie discrètement une requête par ligne au lieu d'une seule jointure.

Les index : l'index du livre, pas le livre entier

Sans aide, la seule façon pour la base de données de trouver toutes les commandes du client 42 est de lire chaque ligne de la table des commandes et de les vérifier une à une : un parcours complet de la table, qui est en O(n) dans la notation vue dans données et algorithmes. Un index de base de données est une structure de données supplémentaire et triée qui associe les valeurs d'une colonne aux lignes qui les contiennent, exactement comme l'index à la fin d'un livre associe des mots à des numéros de page. La plupart des bases de données relationnelles utilisent par défaut un *arbre B* (B-tree), un arbre large, peu profond et toujours équilibré, qui trouve une valeur en O(log n) étapes.

Les clés primaires sont indexées automatiquement. Les autres index sont ajoutés délibérément, pour les colonnes sur lesquelles les requêtes filtrent, font des jointures ou trient : `CREATE INDEX idx_orders_customer ON orders (customer_id);`. Les index ne sont pas gratuits. Chaque insertion, mise à jour ou suppression doit aussi mettre à jour chaque index de la table, et chaque index occupe de l'espace disque et de la mémoire. Une table avec quinze index se lit vite et s'écrit lentement. Choisir les index est un compromis guidé par les requêtes que le système exécute réellement, ce qui est un premier exemple du compromis entre espace et temps qui traverse toute l'architecture.

EXEMPLE COMMENTÉ

Exemple commenté : trouver les commandes d'un client

Une boutique a 10 millions de lignes dans sa table des commandes. L'écran du service client demande toutes les commandes d'un client, soit environ 20 lignes.

Sans index	Parcours complet de la table : la base de données lit et vérifie les 10 000 000 lignes pour trouver les 20 qui correspondent.
Avec un index en arbre B sur customer_id	Chaque page d'index contient des centaines de clés, si bien que l'arbre n'a que trois ou quatre niveaux. La base de données lit une poignée de pages d'index, puis va chercher les 20 lignes correspondantes.
Croissance	Si la table passe à 100 millions de lignes, le parcours fait dix fois plus de travail ; la recherche par l'index ajoute au plus un niveau.
Un coût payé ailleurs	Chaque nouvelle commande écrit désormais dans la table et dans l'index. L'espace disque utilisé augmente de la taille de l'index.

C'est O(n) contre O(log n), vu à la section 0.2 : souvent la différence entre une réponse en quelques secondes et une réponse en quelques millisecondes. Beaucoup de plaintes sur des requêtes lentes dans les jeunes systèmes se règlent avec le bon index, pas avec du matériel plus puissant.

Transactions et ACID : le virement bancaire

Virer 100 livres du compte d'Alice vers celui de Bob demande deux modifications : retirer 100 à Alice, ajouter 100 à Bob. Si le serveur tombe en panne après la première modification et avant la seconde, 100 livres ont disparu. Si deux virements depuis le compte d'Alice s'exécutent en même temps, tous deux peuvent voir un solde de 150 et réussir tous les deux, ce qui la met à découvert. Une transaction regroupe plusieurs opérations en une seule unité de travail qui soit aboutit entièrement (validation, ou *commit*), soit n'a aucun effet (annulation, ou *rollback*).

Les garanties qu'offre une transaction se résument par l'acronyme ACID, créé par Theo Härder et Andreas Reuter en 1983 à partir de travaux antérieurs de Jim Gray :

Propriété	Ce qu'elle promet	Dans le virement bancaire
Atomicité	Toutes les modifications de la transaction ont lieu, ou aucune.	Alice n'est jamais débitée sans que Bob soit crédité.
Cohérence	Une transaction fait passer la base de données d'un état valide à un autre ; les règles comme les clés et les contrôles ne sont jamais laissées enfreintes.	Une règle interdisant aux soldes de passer sous zéro rejette un virement qu'Alice ne peut pas se permettre.
Isolation	Les transactions qui s'exécutent en même temps ne voient pas le travail à moitié fait des autres ; le résultat est le même que si elles s'étaient exécutées l'une après l'autre (dans une mesure fixée par le niveau d'isolation).	Deux virements simultanés depuis le compte d'Alice ne peuvent pas s'appuyer tous les deux sur le même ancien solde.
Durabilité	Une fois que la base de données a confirmé un commit, la modification survit à une panne ou à une coupure de courant.	Quand l'application affiche « Virement effectué », l'argent a vraiment été transféré, car la modification est déjà dans le journal sur disque.

```
import sqlite3

conn = sqlite3.connect("bank.db")
# The accounts table was created with CHECK (balance ⩾ 0)

def transfer(from_id, to_id, amount):
    try:
        with conn:  # one transaction: commit on success, rollback on error
            debit = conn.execute(
                "UPDATE accounts SET balance = balance - ? WHERE id = ?",
                (amount, from_id))
            credit = conn.execute(
                "UPDATE accounts SET balance = balance + ? WHERE id = ?",
                (amount, to_id))
            if debit.rowcount ≠ 1 or credit.rowcount ≠ 1:
                raise ValueError("No such account")  # forces a rollback
        return "Transfer complete"
    except (sqlite3.Error, ValueError):
        return "Transfer refused: no account was changed"
```

Un virement avec le module sqlite3 intégré à Python. Le bloc with valide la transaction si les deux mises à jour réussissent, et l'annule si l'une échoue ou si un compte n'existe pas.

Remarquez la vérification de `rowcount` (le nombre de lignes modifiées par chaque mise à jour) : une mise à jour qui ne trouve aucun compte n'est pas une erreur pour la base de données, donc le code doit lui-même la traiter comme telle. Deux autres détails comptent au-delà de cet

exemple. Les marqueurs `?` transmettent les valeurs séparément du texte SQL, ce qui est la défense standard contre l'*injection SQL*, une attaque où une saisie de l'utilisateur est glissée dans une requête sous forme de code (le chapitre 8 y revient). Et la promesse de durabilité est tenue grâce à un *journal d'écriture anticipée* (write-ahead log) : le SGBD enregistre chaque modification dans un journal sur disque, où l'on ne fait qu'ajouter, avant de confirmer le commit, de sorte qu'après une panne il peut rejouer ou annuler du travail pour revenir à un état correct.

L'isolation est la propriété la plus subtile. L'isolation complète (dite *sérialisable*) est sûre mais limite la quantité de travail qui peut s'exécuter en parallèle ; les bases de données proposent donc des *niveaux d'isolation* plus faibles qui échangent une partie de la protection contre de la vitesse, et beaucoup utilisent par défaut un niveau plus faible. Les anomalies que chaque niveau autorise sont traitées dans transactions et isolation. Pour l'instant, l'idée clé est qu'une base de données unique offre des garanties fortes et bien comprises à l'intérieur d'une transaction, et que ces garanties sont bien plus difficiles à obtenir une fois les données réparties entre plusieurs bases de données ou services.

NoSQL : d'autres formes de données

Les bases de données relationnelles ne sont pas la seule option. NoSQL est un nom générique, popularisé lors d'une rencontre à San Francisco en 2009, pour les bases de données qui n'utilisent pas le modèle relationnel. Elles se sont développées pour traiter de très gros volumes répartis sur de nombreuses machines, des formes de données souples et des modes d'accès que les tables servent mal. Le nom dit ce qu'elles ne sont pas, et il couvre donc des familles très différentes.

Les quatre grandes familles NoSQL

Famille	Comment elle stocke les données	Bien adaptée à	Exemples
Base orientée documents	Chaque enregistrement est un document autonome, en général proche du JSON, qui peut imbriquer des listes et des sous-objets. Les documents d'une même collection n'ont pas besoin d'avoir exactement les mêmes champs.	Les données lues et écrites d'un seul bloc : un catalogue de produits aux attributs variés, un profil utilisateur, le contenu d'une page web.	MongoDB, Couchbase, Firestore
Base clé-valeur	Un dictionnaire géant : une clé unique correspond à une valeur dont la base de données ne regarde pas le contenu. La recherche se fait uniquement par la clé.	Les recherches simples et très rapides : sessions, paniers d'achat, caches, feature flags.	Redis, Amazon DynamoDB, etcd
Base orientée colonnes larges	Des lignes identifiées par une clé, chacune avec de nombreuses colonnes, regroupées et réparties (partitionnées) sur de nombreuses machines. Les tables sont conçues autour des requêtes qui seront exécutées.	D'énormes volumes d'écritures, comme des relevés de capteurs, des journaux d'événements et des historiques de messagerie, sur de nombreux serveurs et centres de données.	Apache Cassandra, Apache HBase, Google Bigtable
Base orientée graphe	Des nœuds (les choses) et des arêtes (les relations), les uns comme les autres avec des propriétés, comme dans les graphes de la section 0.2.	Les questions sur les liens : amis d'amis, réseaux de fraude, calcul d'itinéraires, recommandations.	Neo4j, Amazon Neptune

Les spécialistes ne sont pas d'accord sur le choix par défaut. Un camp conseille de commencer par une base de données relationnelle, sauf raison claire de faire autrement : SQL est très connu, les transactions et les jointures sont puissantes, et les systèmes relationnels modernes stockent aussi très bien les documents JSON. L'autre camp conseille de choisir dès le départ le stockage adapté au mode d'accès, car une base orientée documents ou clé-valeur peut être plus simple et passer à l'échelle à moindre coût pour la bonne charge de travail. Les deux s'accordent sur le fait que beaucoup de systèmes réels utilisent plus d'un type de base, chacun là où il convient : c'est ce qu'on appelle la *persistance polyglotte*. La section du chapitre 7 consacrée au choix d'une base de données donne une méthode structurée pour décider.

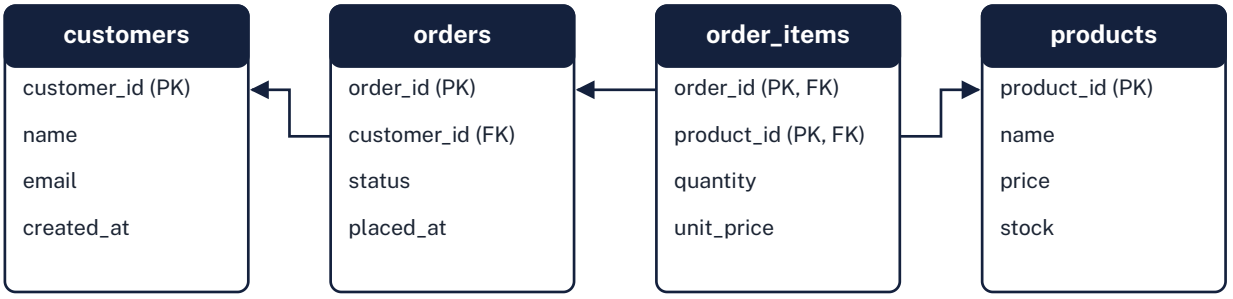
ATTENTION

« Sans schéma » ne veut pas dire « pas de schéma »

Une base orientée documents acceptera aujourd'hui un document client avec `email` et demain avec `emailAddress`. Les données ont toujours une forme ; simplement, celle-ci est désormais imposée, ou non, par chaque programme qui les lit, et non plus par la base de données. Les équipes qui font l'impasse sur la réflexion sur le schéma dans les bases NoSQL la redécouvrent en général plus tard, sous la forme d'un code plein de cas particuliers pour les anciens documents.

Exemple commenté : le schéma d'une boutique en ligne

Prenons une petite boutique en ligne. Elle doit savoir qui sont ses clients, ce qu'elle vend et qui a commandé quoi. Une première tentative pourrait tout mettre dans une seule table, avec une ligne par commande et des colonnes `product1`, `product2`, `product3`. Cela ne marche plus dès que quelqu'un commande quatre articles, et cela recopie le nom et l'adresse du client dans chaque commande.



Chaque flèche va d'une clé étrangère (FK) à la clé primaire (PK) qu'elle référence.

Un schéma de boutique normalisé : chaque fait est stocké une seule fois, et les clés étrangères enregistrent les relations.

EXEMPLE COMMENTÉ

Exemple commenté : pourquoi le schéma a quatre tables

Le schéma ci-dessus répond aux questions de la boutique sans recopier de faits.

customers	Une ligne par personne. Changer une adresse e-mail modifie une seule ligne.
orders	Une ligne par passage au paiement. `customer_id` est une clé étrangère, donc chaque commande appartient à un vrai client. Un client a de nombreuses commandes : c'est une relation *un-à-plusieurs*.
products	Une ligne par article en vente, avec le prix actuel et le niveau de stock.
order_items	Une commande contient de nombreux produits et un produit figure dans de nombreuses commandes : c'est une relation *plusieurs-à-plusieurs*. Les bases de données relationnelles la représentent par une table de liaison, avec une ligne par produit et par commande.
Copie délibérée : unit_price	Le prix est recopié dans `order_items` au moment où la commande est passée. Ce n'est pas une erreur de normalisation : la commande doit garder la trace de ce que le client a réellement payé, même si le prix du catalogue change la semaine suivante.
Passer une commande	L'insertion de la commande, l'insertion de ses articles et la baisse du stock de chaque produit se font dans une seule transaction, si bien qu'une panne ne peut jamais laisser une commande sans articles, ni un stock qui n'a pas été diminué.

Une bonne conception de schéma part des questions que se pose l'entreprise et des faits qu'elle ne doit jamais perdre, pas des écrans de la première version de l'application.

La même boutique, dans une base orientée documents, pourrait stocker chaque commande comme un seul document contenant ses articles imbriqués, ce qui convient pour lire une commande entière d'un coup, mais complique des questions comme « total des ventes par produit ».

Sauvegardes, restauration et bases de données managées

Les disques tombent en panne, des personnes lancent le mauvais `DELETE`, les logiciels ont des bugs et des attaquants chiffrent les données pour obtenir une rançon. Une sauvegarde est une copie des données conservée dans un endroit séparé, à partir de laquelle la base de données peut être restaurée. Trois habitudes professionnelles comptent. Une sauvegarde qui n'a jamais été restaurée est un espoir, pas une sauvegarde : les équipes s'exercent donc à restaurer et chronomètrent l'opération. Les copies sont conservées à plus d'un endroit (une règle empirique courante : trois copies, sur deux types de supports, dont une hors site). Et une *réplique*, une

seconde base de données qui copie la première en continu, protège contre la panne d'une machine, mais pas contre les erreurs : un mauvais `DELETE` atteint la réplique en quelques millisecondes. Beaucoup de bases de données peuvent combiner une sauvegarde complète périodique et le journal des transactions conservé pour revenir à n'importe quel moment : c'est la *restauration à un instant donné* (point-in-time recovery).

Bien exploiter une base de données (appliquer les correctifs, l'optimiser, la répliquer, la sauvegarder et la restaurer à trois heures du matin) est un travail spécialisé et sans fin ; les fournisseurs de cloud vendent donc des bases de données managées : Amazon RDS, Azure SQL Database et Google Cloud SQL en sont des exemples. Le fournisseur s'occupe des machines, des mises à jour logicielles, des sauvegardes automatiques et du basculement (failover) vers une copie de secours ; le client reste responsable du schéma, des requêtes, des index, des droits d'accès et du coût. Les services managés sont traités plus largement dans serveurs et cloud.

Pourquoi c'est important pour l'architecture

Les données survivent en général au code qui les entoure : les applications sont réécrites tous les quelques ans, alors que les dossiers clients et les historiques médicaux sont conservés pendant des décennies. Les décisions sur l'endroit où se trouvent les données, le type de base qui les contient, le programme qui en est propriétaire et les garanties dont elles ont besoin comptent donc parmi les plus coûteuses à remettre en cause. Plusieurs grandes questions d'architecture découlent directement de cette section : chaque service doit-il posséder sa propre base de données (propriété des données), quelle base de données convient à quel usage (choisir une base de données), et comment garder des données correctes quand une seule action métier touche plusieurs bases de données, là où une transaction ACID unique n'est plus possible (sagas). Quand un architecte demande « où se trouve la source de vérité pour ceci ? », c'est à cause de cette section.

VÉRIFIEZ VOTRE COMPRÉHENSION

▼ **Pourquoi un fichier de tableur est-il un mauvais endroit pour garder les commandes d'une boutique en ligne dès que de nombreux clients achètent en même temps ?**

Des écritures simultanées peuvent s'écraser les unes les autres, une panne en cours d'enregistrement peut corrompre le fichier, trouver les commandes d'un client oblige à tout lire, et rien n'impose de règles comme le fait que chaque commande ait un vrai client. Un SGBD résout ces quatre problèmes.

▼ **Dans le schéma de la boutique, qu'est-ce qui fait de customer_id dans la table orders une clé étrangère, et qu'est-ce que cela vous apporte ?**

Elle contient la clé primaire d'une ligne de la table customers et elle est déclarée comme une référence à celle-ci. La base de données refuse alors les commandes pour des clients qui n'existent pas.

▼ **Pourquoi ne pas simplement indexer toutes les colonnes de toutes les tables ?**

Chaque index doit être mis à jour à chaque insertion, mise à jour et suppression, et il occupe de l'espace. Trop d'index rendent les écritures lentes et le stockage coûteux ; on choisit donc les index en fonction des requêtes réellement exécutées.

▼ **Quelle propriété ACID est violée si une panne laisse Alice débitée sans que Bob soit crédité ?**

L'atomicité : les deux modifications auraient dû avoir lieu ensemble, ou pas du tout.

▼ **Une équipe déclare : « Nous avons une réplique dans un autre centre de données, donc nous n'avons pas besoin de sauvegardes. » Qu'est-ce qui lui a échappé ?**

Une réplique copie les erreurs et les données corrompues aussi fidèlement que les bonnes données ; une suppression accidentelle ou une attaque par rançongiciel l'atteint donc en quelques instants. Les sauvegardes, idéalement avec restauration à un instant donné, protègent contre cela.

▼ **À vous : Un théâtre garde ses ventes dans une seule table avec les colonnes name, email, show, seat1, seat2 et seat3. Certains clients veulent désormais quatre places et beaucoup réservent à plusieurs reprises. Quelles tables utiliseriez-vous à la place, et qu'est-ce qui les relie entre elles ?**

Utilisez customers, shows, bookings et booked_seats. Une réservation a une clé étrangère vers son client et vers son spectacle, et chaque place réservée est une ligne avec une clé étrangère vers sa réservation. Quatre places font simplement quatre lignes, et un e-mail n'est stocké qu'une fois : le changer ne touche donc qu'une ligne. Une règle d'unicité sur le spectacle et la place empêche de vendre deux fois la même place, et une transaction rend chaque réservation tout ou rien.

SOURCES

- PostgreSQL documentation: Transactions (tutorial): https://www.postgresql.org/docs/current/tutorial-transactions.html
- PostgreSQL documentation: Indexes, introduction: https://www.postgresql.org/docs/current/indexes-intro.html

- PostgreSQL documentation: Joins between tables: https://www.postgresql.org/docs/current/tutorial-join.html
- Martin Fowler: NosqlDefinition: https://martinfowler.com/bliki/NosqlDefinition.html

SECTION 0.6 SUR 0.8 · DÉBUTANT · 20 MIN DE LECTURE · DERNIÈRE RELECTURE : NOVEMBRE 2026

API, JSON et coopération entre systèmes distincts

L'IDÉE EN BREF

Une API ressemble à la carte d'un restaurant et à son serveur. La carte vous dit exactement ce que vous pouvez commander et ce que vous allez recevoir ; le serveur porte votre commande en cuisine et vous rapporte le plat. Vous n'entrez jamais dans la cuisine, et la cuisine peut changer sa façon de cuisiner sans que vous le remarquiez, tant que les plats de la carte restent les mêmes. Les logiciels utilisent les API de la même manière : un programme demande quelque chose à un autre dans un format convenu et reçoit en retour une réponse convenue.

CE QU'IL FAUT SAVOIR

- Une API est un contrat entre des éléments logiciels : ce qu'on peut demander, avec quelles entrées, et ce qu'on reçoit en retour, tandis que l'implémentation qui se trouve derrière reste cachée et libre de changer.
- Les API internes à un processus sont des appels de fonction à l'intérieur d'un même programme ; les API web traversent un réseau, elles sont donc plus lentes et peuvent échouer d'une façon dont un appel de fonction ne le peut pas.
- Les API REST exposent des ressources à des URL, agissent sur elles avec des méthodes HTTP (GET, POST, PUT, PATCH, DELETE), signalent les résultats par des codes d'état et échangent en général du JSON.
- Les clés d'API identifient une application appelante ; les jetons d'accès sont des identifiants de courte durée, souvent au nom d'un utilisateur ; les uns comme les autres ne doivent circuler que par HTTPS et rester hors du code source.
- Un webhook, c'est le fournisseur qui appelle l'URL du client quand un événement se produit ; le destinataire doit vérifier les signatures et gérer les doublons.
- Les appels synchrones attendent une réponse ; les messages asynchrones passent par une file, si bien que l'émetteur et le destinataire n'ont pas besoin d'être disponibles en même temps.
- Les changements cassants pénalisent les clients, c'est pourquoi les API sont versionnées, et les limites de débit (HTTP 429) protègent les fournisseurs ; toute API tierce est une dépendance qu'il faut isoler et prévoir.

Ce qu'est une API

Une API (interface de programmation d'application) est la façon définie dont un élément logiciel demande à un autre de faire quelque chose. Elle indique ce qu'on peut demander, ce qu'il faut fournir, ce qu'on reçoit en retour et ce qui peut mal se passer. Le mot qui compte est *définie* : une API est un contrat. Le fournisseur promet de la maintenir en état de marche ; le consommateur promet de n'utiliser que ce qu'elle décrit.

Le restaurant est l'analogie habituelle. La carte indique ce que vous pouvez commander et ce que contient chaque plat ; le serveur porte votre commande en cuisine et vous rapporte le résultat. Vous n'entrez pas dans la cuisine, et vous n'avez pas besoin de savoir si le chef cuisine au gaz ou à l'induction. La cuisine peut changer d'équipement, de personnel et de fournisseurs à volonté, tant que les plats de la carte arrivent toujours comme décrit. La carte est l'API, la cuisine est l'implémentation, et c'est la séparation entre les deux qui permet à chaque côté de changer indépendamment de l'autre.

C'est la même idée que les interfaces vues dans organiser le code, appliquée à toutes les échelles. C'est aussi pourquoi les architectes accordent tant d'importance aux API : une API est l'endroit où le travail d'une équipe rencontre celui d'une autre, et dès que d'autres en dépendent, elle coûte cher à modifier.

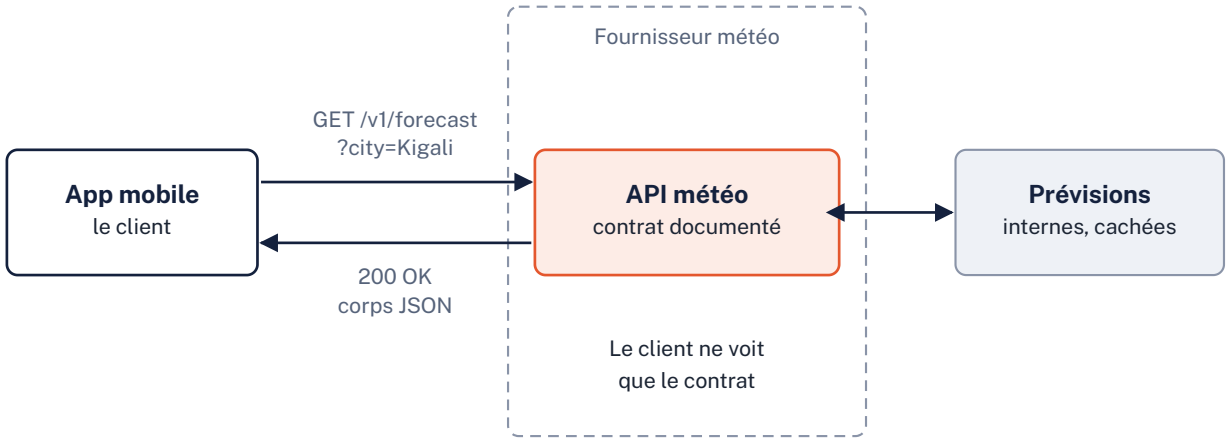
API internes au processus et API web

Certaines API sont appelées à l'intérieur d'un même programme en cours d'exécution. Quand du code Python appelle `json.loads(text)`, il utilise l'API du module `json` de la bibliothèque standard : un appel de fonction qui prend quelques nanosecondes et n'échoue que si l'entrée est incorrecte. Les fonctions et classes publiques d'une bibliothèque forment son API interne au processus.

Une API web est appelée par un réseau, en général avec HTTP (voir comment les ordinateurs communiquent). L'appelant et le fournisseur peuvent se trouver sur des machines différentes, être écrits dans des langages différents et appartenir à des organisations différentes. Une application bancaire mobile parle à l'API web de la banque ; les serveurs de la banque parlent à l'API d'un réseau de cartes ; une boutique parle à l'API d'un prestataire de paiement.

La différence n'est pas superficielle. Un appel réseau est des milliers à des millions de fois plus lent qu'un appel de fonction, et il peut échouer d'une façon dont un appel de fonction ne le peut pas : le réseau perd la requête, le serveur est surchargé, la réponse arrive après que l'appelant a abandonné, ou la requête a réussi mais la réponse s'est perdue, si bien que l'appelant ne peut pas savoir si le paiement est passé. Une grande partie de la conception des systèmes distribués, traitée au chapitre 6, consiste à vivre avec ces défaillances.

Un appel d'API web



Le client ne voit que le contrat de l'API ; la façon dont le fournisseur stocke et calcule les prévisions reste cachée derrière lui.

REST : ressources, URL et verbes

La plupart des API web suivent aujourd'hui le style appelé REST (Representational State Transfer), décrit par Roy Fielding dans sa thèse de doctorat de 2000. Fielding a défini REST comme un ensemble de contraintes pour les grands systèmes en réseau comme le web ; dans l'usage courant, « une API REST » désigne quelque chose de plus simple : une API HTTP organisée autour de **ressources**.

Une **ressource** est une chose que l'API expose : un client, une commande, une prévision. Chacune a une URL, comme `/orders/1042`. Une collection de ressources a aussi une URL, comme `/orders`. Le client agit sur les ressources avec les méthodes HTTP standard, lit le code d'état du résultat et échange des données dans le corps, en général au format JSON. Les mêmes quelques verbes fonctionnent donc pour toutes les ressources, ce qui rend prévisibles les API REST bien conçues.

Une ressource, les verbes standard

Requête	Signification	Statut habituel en cas de succès
`GET /orders?customer=42`	Lister les commandes, filtrées par un paramètre de requête. Les lectures ne modifient jamais rien.	200 OK
`GET /orders/1042`	Récupérer une commande.	200 OK (ou 404 Not Found si cette commande n'existe pas)
`POST /orders`	Créer une nouvelle commande à partir du JSON contenu dans le corps de la requête.	201 Created, avec l'URL de la nouvelle commande
`PUT /orders/1042`	Remplacer la commande 1042 par la représentation envoyée.	200 OK ou 204 No Content
`PATCH /orders/1042`	Modifier seulement certains champs, comme l'adresse de livraison.	200 OK
`DELETE /orders/1042`	Annuler ou supprimer la commande.	204 No Content

Les codes d'état portent un sens sur lequel un programme client peut agir. Un code **4xx** indique que le client a fait quelque chose de travers : 400 pour une requête mal formée, 401 quand les identifiants manquent ou ne sont pas valides, 403 quand l'appelant est connu mais n'a pas l'autorisation, 404 quand la ressource n'existe pas, 429 quand il a envoyé trop de requêtes. Un code **5xx** indique que le serveur a échoué : 500 pour une erreur inattendue, 503 quand il est temporairement indisponible. La distinction compte, car renvoyer telle quelle une requête qui a reçu un 4xx échouera généralement de nouveau (408 et 429 sont les exceptions : attendez, puis réessayez), alors que renvoyer une requête qui a reçu un 503 un peu plus tard peut réussir.

Les données circulent au format JSON (JavaScript Object Notation), un format texte normalisé par la RFC 8259. Il n'a que quelques éléments de base : des objets (des paires nom-valeur entre accolades), des tableaux (des listes entre crochets), des chaînes entre guillemets doubles, des nombres, `true`, `false` et `null`. Malgré son nom, tous les langages courants savent le lire et l'écrire, et c'est pourquoi il est devenu la langue commune des API web. D'autres styles existent et répondent à d'autres besoins : GraphQL permet au client de préciser exactement les champs qu'il veut, et gRPC utilise un format binaire compact pour des appels rapides entre services ; le chapitre 6 les compare.

EXEMPLE COMMENTÉ

Exemple commenté : appeler une API météo

Une application de voyage veut les prévisions de demain pour Kigali. Le fournisseur (ici un api.example.com imaginaire) documente un *endpoint* (une URL qui accepte un type de requête précis) et attribue une clé d'API à l'application.

1. Lire le contrat	La documentation indique : `GET /v1/forecast` avec les paramètres de requête `city` et `days` ; envoyer la clé dans un en-tête `Authorization` ; réponses en JSON ; limite de 60 requêtes par minute.
2. Envoyer la requête	L'application envoie la requête HTTP présentée ci-dessous, par HTTPS, pour que la clé et les données soient chiffrées pendant leur transport.
3. Vérifier le statut	200 signifie que le corps contient une prévision. 401 signifie que la clé est fausse ; 429 signifie qu'il faut ralentir ; 503 signifie qu'il faut réessayer plus tard.
4. Analyser le JSON	L'application lit `days[0].max_c` et `days[0].rain_chance` et ignore les champs dont elle n'a pas besoin.
5. Gérer l'échec	Si aucune réponse n'arrive dans un délai de, disons, 3 secondes, l'application affiche la dernière prévision mise en cache plutôt qu'un écran vide.

Chaque étape dépend du contrat, pas de la façon dont le fournisseur calcule les prévisions. Le fournisseur peut passer demain à de nouveaux serveurs ou à un nouveau modèle météo sans que l'application change.

L'étape 5 est celle que les débutants oublient : tout appel à travers un réseau peut être lent ou échouer, l'appelant doit donc décider quoi faire quand cela arrive.

```
GET /v1/forecast?city=Kigali&days=2 HTTP/1.1
Host: api.example.com
Authorization: Bearer sk_test_4f9a...
Accept: application/json

HTTP/1.1 200 OK
Content-Type: application/json

{
  "city": "Kigali",
  "units": "metric",
  "days": [
    {"date": "2026-10-23", "min_c": 16, "max_c": 27, "rain_chance": 0.6},
    {"date": "2026-10-24", "min_c": 16, "max_c": 26, "rain_chance": 0.4}
  ]
}
```

La requête et la réponse de l'exemple commenté.

```
import os
import requests

API_KEY = os.environ["WEATHER_API_KEY"]   # never hard-code secrets

resp = requests.get(
    "https://api.example.com/v1/forecast",
    params={"city": "Kigali", "days": 2},
    headers={"Authorization": f"Bearer {API_KEY}"},
    timeout=3,                # give up after 3 seconds
)
resp.raise_for_status()       # raises an error for 4xx and 5xx
tomorrow = resp.json()["days"][0]
print(tomorrow["max_c"], "degrees, rain chance", tomorrow["rain_chance"])
```

Le même appel en Python, avec la bibliothèque requests, très répandue, y compris un délai d'expiration (timeout) et la vérification du statut.

Qui appelle ? Clés d'API et jetons

La plupart des API ont besoin de savoir qui appelle, pour facturer, limiter et décider de ce que l'appelant peut voir. Deux idées méritent d'être distinguées dès le départ. L'*authentification* répond à la question « qui êtes-vous ? » ; l'*autorisation* répond à « qu'avez-vous le droit de faire ? »

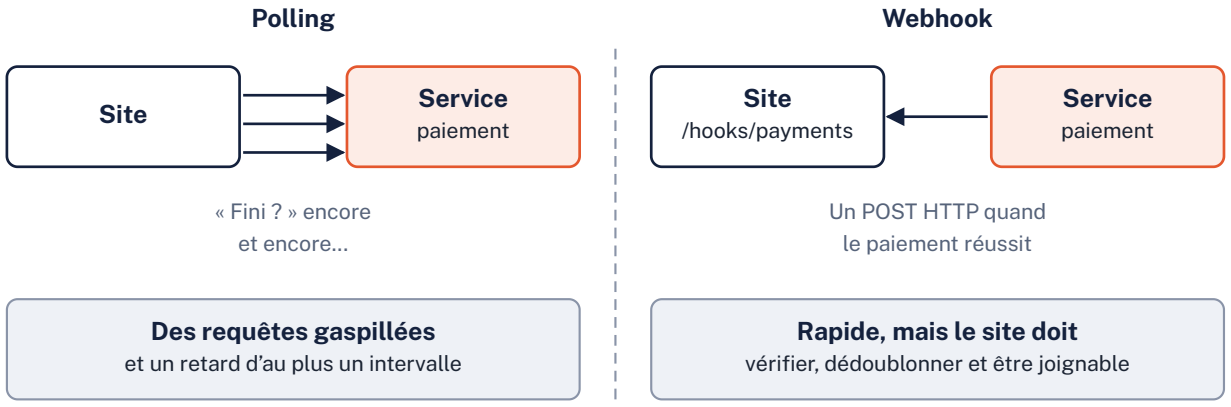
Une clé d'API est une longue chaîne secrète qui identifie l'application appelante ; elle est envoyée avec chaque requête, en général dans un en-tête. C'est simple, et cela identifie un programme plutôt qu'une personne. Un jeton d'accès (access token) est un identifiant de courte

durée délivré après la connexion d'une personne, souvent au nom d'un utilisateur précis et avec des droits précis (« lire l'agenda de cet utilisateur, mais pas ses e-mails ») ; il est en général envoyé sous la forme `Authorization: Bearer <token>`. Les jetons expirent, si bien qu'un jeton volé n'est utilisable que pendant un temps limité. Les protocoles standard qui les sous-tendent, OAuth 2.0 et OpenID Connect, sont traités dans identité et accès.

Quel que soit le moyen utilisé, les règles sont les mêmes : n'envoyer les identifiants que par HTTPS, les garder hors du code source et hors des applications mobiles, d'où n'importe qui peut les extraire, les stocker dans un gestionnaire de secrets ou dans les paramètres d'environnement, et pouvoir les révoquer et les remplacer rapidement.

Webhooks : l'API vous rappelle

Parfois, le client a besoin de savoir quand quelque chose se produit du côté du fournisseur : un paiement réussit, un colis est livré, un document est signé. Le client pourrait faire du *polling*, c'est-à-dire demander « est-ce que c'est fait ? » toutes les quelques secondes, ce qui gaspille des requêtes et repère quand même l'événement en retard. Un webhook inverse le sens : le client enregistre une URL qui lui appartient, et le fournisseur envoie un POST HTTP à cette URL quand l'événement se produit. Le terme a été inventé par Jeff Lindsay en 2007.



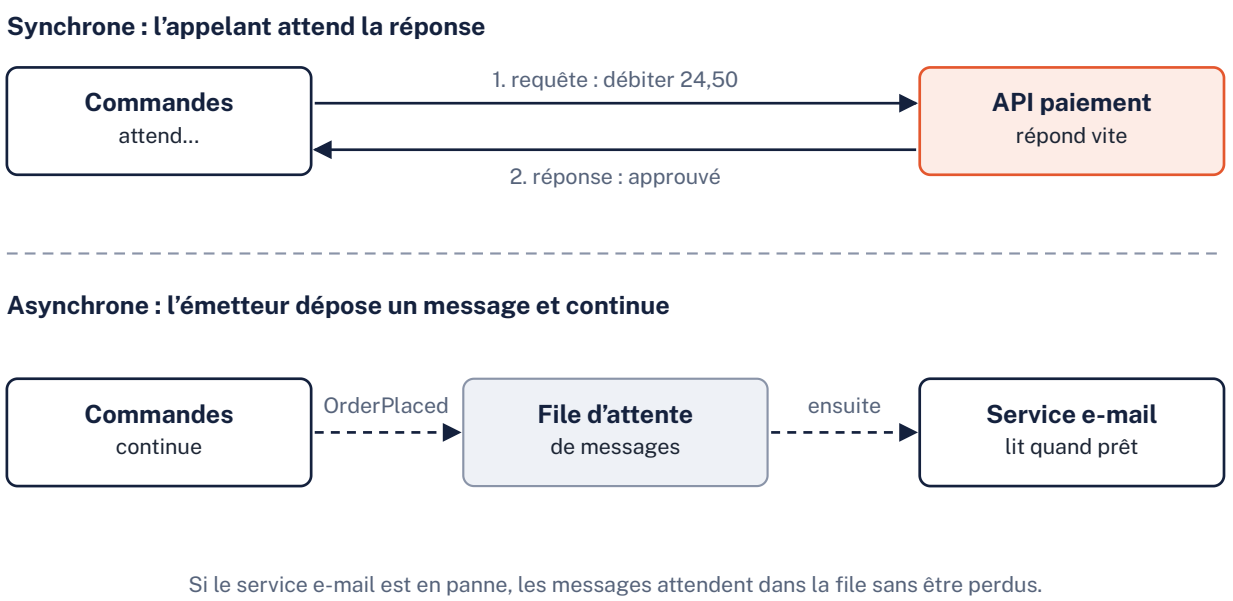
Le polling demande sans cesse ; un webhook attend qu'on le prévienne, au prix de nouvelles responsabilités pour le destinataire.

Les webhooks apportent des responsabilités. L'endpoint destinataire est sur l'internet public, il doit donc vérifier que chaque appel vient bien du fournisseur, en général en vérifiant une signature cryptographique placée dans un en-tête. Les fournisseurs réessaient quand le destinataire ne répond pas, si bien qu'un même événement peut arriver deux fois et que le destinataire doit reconnaître les doublons (traiter un doublon ne doit pas expédier la commande deux fois). Et les événements peuvent arriver dans le désordre.

Attendre une réponse, ou non : synchrone et asynchrone

Jusqu'ici, tout était synchrone : l'appelant envoie une requête et attend la réponse avant de continuer, comme lors d'un appel téléphonique. C'est simple à comprendre, et c'est le bon choix quand l'appelant a vraiment besoin de la réponse tout de suite, par exemple « ce paiement par carte a-t-il été accepté ? ». Sa faiblesse est que l'appelant n'est jamais plus disponible ni plus rapide que tout ce qu'il attend.

L'autre possibilité est la messagerie asynchrone, qui ressemble plutôt à une lettre déposée dans une boîte aux lettres. L'émetteur place un message dans une file de messages et poursuit son travail. Le destinataire prend les messages dans la file quand il est prêt. Si le destinataire est lent ou en panne pendant une heure, les messages attendent au lieu d'échouer. Quand la boutique confirme une commande, l'envoi de l'e-mail de confirmation n'a pas besoin de bloquer l'écran du client ; un message `OrderPlaced` peut aller dans une file, et le service d'e-mails le traitera un instant plus tard.



Un appel synchrone lie l'appelant à la disponibilité de l'appelé ; une file les découple dans le temps.

L'asynchronisme n'est pas gratuit. L'émetteur n'apprend pas tout de suite si le travail a réussi, le système devient plus difficile à tracer et à tester, et les messages peuvent être livrés plus d'une fois. Décider quelles interactions doivent être synchrones et lesquelles asynchrones est l'une des décisions centrales des systèmes distribués, traitée en détail dans communication et API et dans files et contre-pression.

GUIDE DE DÉCISION

Guide de décision : appeler tout de suite ou envoyer un message ?

Un premier guide pour une interaction entre deux parties d'un système. Le chapitre 6 l'affine.

Q1 **L'appelant a-t-il besoin du résultat avant de pouvoir répondre à son propre utilisateur (par exemple, la carte a-t-elle été acceptée) ?**

- **Oui** Utilisez un appel synchrone, avec un délai d'expiration (timeout) et un plan pour le cas où l'autre côté est lent ou en panne.
- **Non** Passez à la question suivante.

Q2 **Le travail peut-il se faire quelques secondes ou quelques minutes plus tard sans que personne ne le remarque (envoyer un e-mail, mettre à jour un rapport) ?**

- **Oui** Envoyez un message asynchrone par une file, pour qu'un destinataire lent ou en panne ne bloque pas l'appelant.
- **Non** Passez à la question suivante.

Q3 **Le travail est-il lent (de quelques secondes à plusieurs heures), comme la génération d'un gros export ?**

- **Oui** Acceptez la requête tout de suite (par exemple avec 202 Accepted), faites le travail à partir d'une file, et prévenez le client quand c'est terminé, par un webhook ou une URL de statut qu'il peut consulter.
- **Non** Un simple appel synchrone convient en général ; gardez-le rapide et fixez un délai d'expiration.

Vivre avec les API des autres : SDK, versions et limites

Beaucoup de fournisseurs publient un SDK (kit de développement logiciel) : une bibliothèque pour chaque langage courant, qui enveloppe les appels HTTP dans des fonctions ordinaires, si bien qu'un développeur écrit `client.refunds.create(payment_id)` au lieu de construire les requêtes à la main. Les SDK font gagner du temps et gèrent des détails comme les nouvelles tentatives et les signatures, mais ils sont une dépendance de plus à tenir à jour.

Dès qu'une API a des utilisateurs, la modifier devient délicat. Ajouter un nouveau champ facultatif à une réponse est en général sans danger. Renommer un champ, en supprimer un, changer son type ou son sens, ou rendre obligatoire une entrée facultative est un changement cassant (breaking change) : chaque client qui comptait sur l'ancien comportement casse, souvent en production, et souvent chez un fournisseur qui ne peut ni voir ni mettre à jour ces clients (pensez aux applications mobiles que les utilisateurs ne mettent jamais à jour). Les

fournisseurs versionnent donc leurs API, par exemple avec `/v1/` et `/v2/` dans l'URL ou avec un en-tête de version, maintiennent l'ancienne version pendant une durée annoncée et publient bien à l'avance des *dépréciations* (des avis indiquant qu'une fonctionnalité va être retirée). Une observation de Hyrum Wright, connue sous le nom de loi de Hyrum, ajoute une mise en garde : avec suffisamment d'utilisateurs, quelqu'un finira par dépendre de chaque comportement observable d'une API, y compris de ceux que la documentation n'a jamais promis.

Les fournisseurs se protègent aussi avec une limite de débit (rate limit) : un plafond comme 100 requêtes par minute et par clé. Les requêtes au-delà de la limite sont refusées avec le statut 429 Too Many Requests, souvent accompagné d'un en-tête `Retry-After` qui indique quand réessayer. Les clients doivent la respecter, étaler leurs requêtes, mettre en cache les réponses qui changent rarement et ralentir quand on le leur demande.

Utiliser des API tierces : développer ou acheter

Les applications modernes sont autant assemblées qu'écrites. Une application de livraison de repas peut utiliser l'API d'un prestataire de paiement pour encaisser les paiements par carte, une API de cartographie pour les adresses et les itinéraires, une API d'e-mail ou de SMS pour les reçus, et un service de notifications push pour « votre livreur est tout près ». Chacune fait gagner des mois de travail spécialisé et apporte une expertise (sécurité des cartes, données cartographiques mondiales, délivrabilité des e-mails) que peu d'équipes pourraient égaler.

Chacune apporte aussi une dépendance que l'équipe ne contrôle pas : les pannes du fournisseur deviennent vos pannes, ses changements de prix deviennent vos coûts, sa façon de traiter les données entre dans votre conformité, et le quitter peut obliger à réécrire du code. Les professionnels gèrent cela délibérément. Ils isolent chaque fournisseur derrière une petite interface qui leur est propre, pour qu'une seule partie du code connaisse l'API du fournisseur ; ils fixent des délais d'expiration et décident de ce qui se passe quand le fournisseur est en panne ; ils surveillent la page d'état et le journal des modifications du fournisseur ; et ils notent ce qu'il faudrait faire pour en changer.

Pourquoi c'est important pour l'architecture

L'architecture porte en grande partie sur les parties d'un système et les contrats entre elles, et les API sont ces contrats rendus concrets. Une API facile à bien utiliser et difficile à casser permet aux équipes de travailler en parallèle et de modifier librement leur fonctionnement interne ; une API qui fuit ou qui est instable couple tout le monde à tout le monde. Le choix entre appels synchrones et messages asynchrones détermine la façon dont les défaillances se propagent. Le chapitre 3 explique les principes des bonnes interfaces (masquage de l'information et abstraction), et le chapitre 6 montre comment les choix d'API et de messagerie font ou défont les systèmes distribués.

VÉRIFIEZ VOTRE COMPRÉHENSION

▼ Dans l'analogie du restaurant, qu'est-ce qui correspond à l'API, et qu'est-ce qui correspond à l'implémentation ?

La carte (et la façon dont le serveur prend les commandes) est l'API : la liste convenue de ce qu'on peut demander et de ce qu'on reçoit en retour. La cuisine est l'implémentation, libre de changer tant que les promesses de la carte sont tenues.

▼ Quelle est la différence entre une réponse 404 et une réponse 503, et laquelle vaut la peine d'être réessayée plus tard ?

404 Not Found indique que la ressource demandée n'existe pas, un problème côté client qu'une nouvelle tentative ne réglera pas. 503 Service Unavailable indique que le serveur est temporairement incapable de répondre, donc une nouvelle tentative après un délai peut réussir.

▼ Pourquoi le destinataire d'un webhook doit-il être prêt à recevoir deux fois le même événement ?

Les fournisseurs réessaient quand ils ne reçoivent pas à temps une réponse de succès, même si la première livraison a en fait été traitée ; les doublons sont donc normaux. Le destinataire doit reconnaître un événement qu'il a déjà traité et ne pas agir de nouveau.

▼ Ajouter un nouveau champ facultatif à une réponse JSON est-il normalement un changement cassant ?

En général non, car les clients bien conçus ignorent les champs qu'ils ne reconnaissent pas. Supprimer ou renommer un champ, ou changer son type, est un changement cassant.

▼ À vous : Votre app de livraison de repas affiche les distances grâce à l'API d'une société de cartographie, limitée à 60 requêtes par minute. À l'heure du dîner, vous en envoyez 200 et recevez de nombreuses réponses 429. Un collègue propose de réessayer immédiatement. Que recommandez-vous plutôt ?

Réessayer aussitôt aggrave la situation, car un 429 signifie que vous dépassez la limite. Conservez les résultats dans votre propre cache, car la distance entre un restaurant et un code postal change rarement, ce qui réduit fortement les appels. Appliquez un backoff lorsque vous recevez effectivement un 429, et placez le prestataire derrière votre propre module, afin que l'app puisse afficher la page sans distance si l'API tombe en panne. Ne demandez une limite plus élevée que si les chiffres restent insuffisants.

SOURCES

- Roy Fielding (2000), Architectural Styles and the Design of Network-based Software Architectures, Chapter 5: REST: https://ics.uci.edu/~fielding/pubs/dissertation/rest_arch_style.htm
- RFC 8259: The JavaScript Object Notation (JSON) Data Interchange Format: https://www.rfc-editor.org/rfc/rfc8259
- MDN Web Docs: 429 Too Many Requests: https://developer.mozilla.org/en-US/docs/Web/HTTP/Reference/Status/429
- Stripe documentation: Receive events in your webhook endpoint: https://docs.stripe.com/webhooks
- Hyrum's Law: https://www.hyrumslaw.com/

SECTION 0.7 SUR 0.8 · DÉBUTANT · 21 MIN DE LECTURE · DERNIÈRE RELECTURE : NOVEMBRE 2026

Serveurs, systèmes d'exploitation, machines virtuelles, conteneurs et services cloud

L'IDÉE EN BREF

Un serveur n'est qu'un ordinateur dont le travail est de répondre à des requêtes ; il est installé dans un grand bâtiment rempli d'autres serveurs, appelé centre de données. Plutôt que d'acheter leurs propres machines, la plupart des organisations louent désormais de la puissance de calcul à des fournisseurs de cloud, un peu comme les foyers achètent l'électricité au réseau au lieu de faire tourner un groupe électrogène. Vous pouvez louer un ordinateur virtuel complet, une plateforme toute prête pour votre code ou un logiciel fini, et ne payer que ce que vous utilisez. Plus vous louez de choses toutes prêtes, moins vous avez à entretenir, et moins vous gardez la main.

CE QU'IL FAUT SAVOIR

- Un serveur est tout ordinateur, ou programme, qui attend des requêtes et y répond ; la plupart des serveurs d'internet fonctionnent sous Linux et se trouvent dans des centres de données conçus pour l'alimentation électrique, le refroidissement et la sécurité.
- Un hyperviseur permet à une seule machine physique de faire tourner plusieurs machines virtuelles, chacune avec son propre système d'exploitation invité, isolée des autres.
- Un conteneur emballe une application avec ses bibliothèques et ses réglages et s'exécute comme un processus isolé qui partage le noyau de l'hôte ; il est donc plus léger et plus rapide à démarrer qu'une VM, mais moins fortement isolé.
- Le cloud computing (NIST SP 800-145) désigne un accès à la demande, en libre-service et payé à l'usage, à des ressources mutualisées et élastiques via le réseau.
- L'IaaS, le PaaS, le serverless (FaaS) et le SaaS confient au fournisseur une part de plus en plus grande de la pile, en échangeant du contrôle contre moins de travail d'exploitation ; le client reste toujours responsable de ses données et de sa configuration.
- Les régions sont des zones géographiques isolées ; les zones de disponibilité sont des groupes de centres de données distincts au sein d'une région ; les CDN servent le contenu statique depuis des emplacements en périphérie proches des utilisateurs.
- La scalabilité verticale (scaling up) consiste à prendre une machine plus grosse ; la scalabilité horizontale (scaling out) consiste à ajouter des machines derrière un équilibreur de charge, ce qui permet d'aller plus loin et de survivre aux pannes, mais seulement si le logiciel est conçu pour cela.

Un serveur n'est qu'un ordinateur qui sert

Un serveur est un ordinateur dont le travail consiste à attendre des requêtes et à y répondre : pages web, appels d'API, requêtes de base de données, fichiers. Le mot désigne un rôle, pas un type particulier de machine. Un ordinateur portable qui fait tourner une petite application web est un serveur tant qu'il répond à des requêtes ; une machine montée en baie dans un centre de données est un serveur parce qu'elle ne fait jamais rien d'autre. Le même mot désigne aussi le programme qui sert (un « serveur web » comme nginx) : le contexte compte donc.

Le matériel d'un serveur comporte les mêmes éléments que n'importe quel ordinateur (voir ce qu'est un logiciel) : processeurs, mémoire, stockage et cartes réseau. La différence tient aux priorités. Les serveurs ont généralement de nombreux cœurs de processeur, beaucoup de mémoire, des alimentations et des disques redondants pour qu'une pièce défaillante n'arrête pas la machine, une administration à distance pour que personne n'ait besoin d'être dans la salle, et ni écran ni clavier. Ils sont conçus pour fonctionner pendant des années sans être éteints.

La plupart des serveurs d'internet fonctionnent sous Linux ou un autre système d'exploitation de type Unix ; Windows Server est courant dans les organisations qui s'appuient sur les produits Microsoft. Un système d'exploitation de serveur remplit les mêmes tâches que celui de votre ordinateur portable (exécuter des processus, gérer la mémoire et les fichiers, communiquer avec le réseau), mais il est généralement administré en ligne de commande, par une connexion à distance, et configuré pour pouvoir être reconstruit automatiquement plutôt qu'à la main.

Les centres de données : là où vivent les serveurs

Les serveurs sont installés dans un centre de données : un bâtiment conçu autour de leurs besoins. Des milliers de machines y sont rangées dans des baies, refroidies par air ou par liquide, car elles transforment presque toute leur électricité en chaleur. Le courant arrive par plusieurs alimentations, secourues par des batteries pendant les premières secondes d'une coupure, puis par des groupes électrogènes diesel. Les connexions réseau proviennent de plusieurs opérateurs. Les portes exigent un badge, et souvent une empreinte digitale.

Une organisation peut exploiter son propre centre de données ou louer de l'espace en baie dans celui de quelqu'un d'autre (on parle de *colocation*) ; avant 2006 environ, c'était presque le seul choix possible. Exploiter soi-même ses serveurs (*sur site*) signifie acheter des machines des mois avant d'en avoir besoin, deviner combien il en faudra, les payer qu'elles soient occupées ou inactives, et employer des personnes pour remplacer les disques en panne en pleine nuit. Ce coût et ce manque de souplesse sont le problème que le cloud a voulu résoudre.

Les machines virtuelles : plusieurs ordinateurs dans un seul

Un serveur typique reste la plupart du temps inactif s'il ne fait tourner qu'une seule application. Une machine virtuelle (VM) permet à une machine physique de se faire passer pour plusieurs. Chaque VM se croit un ordinateur complet, avec ses propres processeurs, sa mémoire, son disque et sa carte réseau, et exécute son propre système d'exploitation, appelé *invité*. Une couche logicielle appelée hyperviseur répartit le matériel réel entre les VM et les tient séparées, de sorte qu'un plantage ou une attaque dans l'une n'atteint pas les autres. Certains hyperviseurs s'exécutent directement sur le matériel (VMware ESXi, Microsoft Hyper-V, KVM, qui fait partie de Linux) ; d'autres s'exécutent comme une application sur un système d'exploitation ordinaire (VirtualBox).

La virtualisation a changé l'économie des serveurs. Une seule machine peut accueillir de nombreuses VM appartenant à différentes équipes ou à différents clients. Une VM est un ensemble de fichiers : on peut donc la copier, la sauvegarder, la déplacer vers une autre machine et la créer en quelques minutes, au lieu de la commander et d'attendre sa livraison. Notez qu'il s'agit d'un autre sens de « machine virtuelle » que les VM de langage, comme la Java Virtual Machine évoquée dans ce qu'est un logiciel : les VM de langage exécutent des programmes, celles dont nous parlons ici imitent des ordinateurs entiers.

Les conteneurs : livrer l'application avec tout ce dont elle a besoin

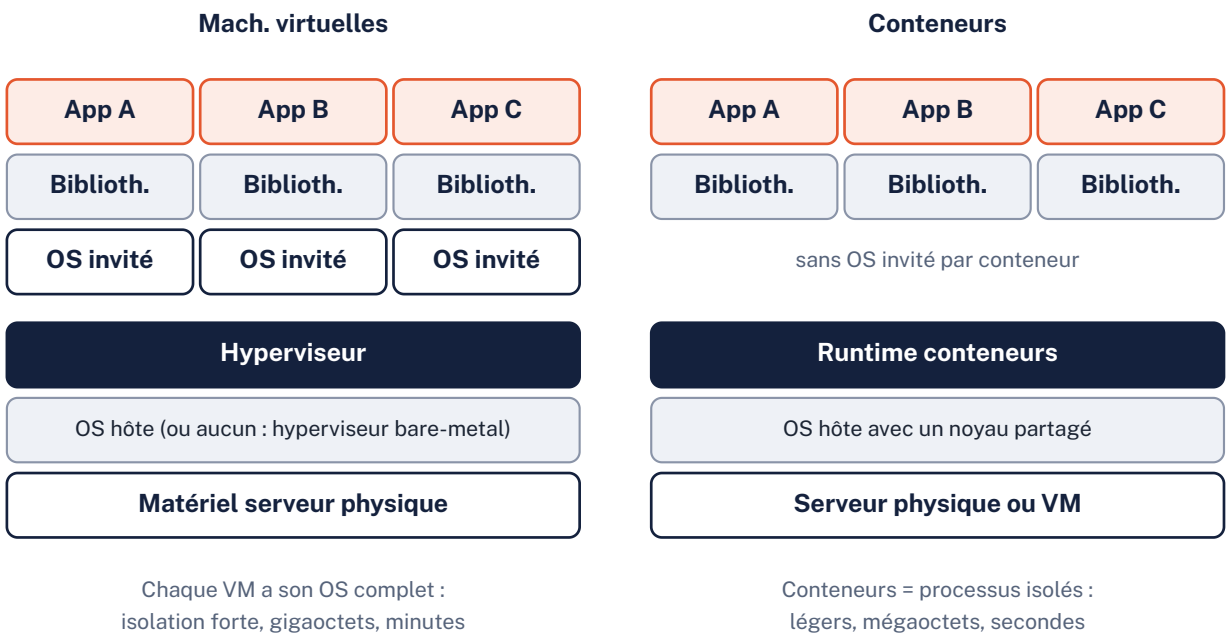
Tout développeur a déjà entendu « ça marche sur ma machine ». Le programme dépend d'une version précise du langage, de bibliothèques précises et de réglages précis, et le serveur en a de légèrement différents. Un conteneur résout ce problème en emballant une application avec tout ce dont elle a besoin pour s'exécuter (son environnement d'exécution, ses bibliothèques, sa configuration) dans une unité standard qui fonctionne de la même façon sur un ordinateur portable, sur un serveur de test ou dans le cloud. Le nom vient des conteneurs maritimes : des boîtes standard que n'importe quelle grue, n'importe quel navire ou camion peut manipuler sans savoir ce qu'elles contiennent.

Le paquet est une image de conteneur : un fichier en lecture seule et versionné, construit à partir d'une courte recette (avec Docker, un `Dockerfile`). Une copie en cours d'exécution d'une image est un conteneur. Docker, sorti en 2013, a rendu cela praticable pour les développeurs au quotidien, et son format d'image est aujourd'hui un standard ouvert maintenu par l'Open Container Initiative, si bien que d'autres outils peuvent construire et exécuter les mêmes images.

La différence essentielle avec une VM tient à ce qui est partagé. Une VM imite du matériel et exécute un système d'exploitation invité complet. Un conteneur est un processus isolé : il partage avec d'autres conteneurs le *noyau* du système d'exploitation de l'hôte (le cœur du

système d'exploitation, qui contrôle le matériel, la mémoire et les processus), et les fonctions d'isolation du noyau lui donnent sa propre vue des fichiers, des processus et du réseau. Les conteneurs démarrent donc en quelques secondes ou moins et occupent des mégaoctets plutôt que des gigaoctets : des dizaines peuvent tourner là où quelques VM tiendraient. La contrepartie est une isolation plus faible, car tous les conteneurs dépendent d'un seul noyau partagé ; c'est pourquoi les fournisseurs de cloud font couramment tourner les conteneurs de leurs clients à l'intérieur de VM, combinant ainsi les deux.

Comparaison des machines virtuelles et des conteneurs



Les VM virtualisent le matériel et chacune embarque un système d'exploitation ; les conteneurs virtualisent le système d'exploitation et en partagent le noyau.

```
# Start from an image that already has Python installed
FROM python:3.12-slim
WORKDIR /app
# Bake the libraries into the image
COPY requirements.txt .
RUN pip install -r requirements.txt
# Add the application code
COPY . .
EXPOSE 8000
CMD ["python", "-m", "uvicorn", "main:app", "--host", "0.0.0.0", "--port", "8000"]
```

Un Dockerfile complet pour une petite API web en Python. La commande docker build transforme cette recette en image.

Faire tourner un conteneur est facile. En faire tourner des centaines sur des dizaines de machines, redémarrer ceux qui plantent, ajouter des copies quand le trafic augmente et déployer de nouvelles versions sans interruption de service ne l'est pas. Ce travail revient à un orchestrateur, dont Kubernetes est le plus connu ; il est traité dans conteneurs et orchestration.

Le cloud : louer de la puissance de calcul à la minute

Le cloud computing (informatique en nuage) consiste à louer des ressources informatiques via internet, à la demande, auprès d'un fournisseur qui exploite d'immenses centres de données : Amazon Web Services (AWS), Microsoft Azure et Google Cloud sont les plus grands, parmi beaucoup d'autres. Le National Institute of Standards and Technology (NIST) américain a publié en 2011 la définition que la plupart des gens citent, la Special Publication 800-145. Elle énumère cinq caractéristiques essentielles : le libre-service à la demande (vous créez vous-même les ressources, via un site web ou une API, sans téléphoner à personne), un large accès par le réseau, la mutualisation des ressources (de nombreux clients partagent le matériel du fournisseur), l'élasticité rapide (la capacité augmente et diminue vite) et le service mesuré (vous payez ce que vous utilisez).

L'analogie, c'est l'électricité. Autrefois, les usines faisaient tourner leurs propres générateurs ; aujourd'hui, elles se branchent sur le réseau et paient au kilowattheure. Avec le cloud, une start-up peut louer cent serveurs pour un après-midi afin de tester un lancement, et cesser de payer dès qu'elle les rend. C'est le paiement à l'usage (pay-as-you-go). Il transforme un gros achat initial en coût de fonctionnement, et il récompense les systèmes capables d'éteindre de la capacité quand elle n'est pas nécessaire. Il crée aussi un nouveau risque : des coûts qui grimpent sans bruit, car rien n'empêche quiconque de créer des ressources puis de les oublier.

IaaS, PaaS, SaaS et serverless : quelle part gérez-vous ?

Les services cloud se distinguent surtout par la part de la pile que le fournisseur gère pour vous. Le NIST a défini trois modèles de service ; un quatrième, le serverless, s'est répandu après le lancement du service Lambda d'AWS en 2014. Une façon courante de les retenir passe par la pizza : vous pouvez la faire maison, en acheter une toute prête à cuire, vous en faire livrer une ou aller au restaurant. Chaque étape confie davantage de travail à quelqu'un d'autre, et vous retire une partie de vos choix.

Comparaison des modèles de service cloud

Modèle	Vous gérez	Le fournisseur gère	Exemples	Analogie de la pizza
Sur site	Tout : bâtiment, matériel, OS, runtime, application, données	Rien	Vos propres serveurs dans votre propre centre de données	Faite maison de A à Z
IaaS (infrastructure en tant que service)	Système d'exploitation, runtime, application, données	Matériel, virtualisation, réseau, bâtiment	Amazon EC2, Azure Virtual Machines, Google Compute Engine	Pizza toute prête à cuire : vous la cuisez dans votre propre four
PaaS (plateforme en tant que service)	Code de l'application et données	Tout ce qui se trouve en dessous, y compris les correctifs de l'OS et la mise à l'échelle	Heroku, Azure App Service, Google App Engine	Livraison : vous fournissez la table et les boissons
Serverless / FaaS (fonction en tant que service)	Des fonctions individuelles, déclenchées par des événements	Tous les serveurs, avec réduction à zéro en cas d'inactivité et facturation à l'appel	AWS Lambda, Azure Functions, Google Cloud Run functions	Pizza à la part, payée à la bouchée
SaaS (logiciel en tant que service)	Vos données, vos utilisateurs et leurs réglages	L'application tout entière	Microsoft 365, Gmail, Salesforce	Manger au restaurant

« Serverless » ne veut pas dire qu'il n'y a pas de serveurs ; cela veut dire que vous ne les voyez et ne les gérez jamais. Vous envoyez une fonction, vous indiquez quels événements la déclenchent (une requête HTTP, l'arrivée d'un fichier, un message dans une file) et le fournisseur exécute autant de copies que nécessaire, en facturant à l'exécution et à la milliseconde. C'est bon marché pour un trafic faible ou en pics, et cela ne demande presque aucun travail d'exploitation. Ses coûts apparaissent ailleurs : le premier appel après une période calme peut être lent, le temps que le fournisseur démarre la fonction (un *démarrage à froid*, ou cold start), la durée d'exécution d'une fonction est limitée, et les coûts peuvent dépasser ceux d'un simple serveur pour un trafic important et régulier. Le chapitre 6 traite en profondeur des architectures serverless.

Entre ces modèles se trouvent les services managés : bases de données, files de messages, caches, stockage objet, moteurs de recherche et bien d'autres, que le fournisseur installe, exploite, corrige, sauvegarde et met à l'échelle. Utiliser une base de données managée au lieu d'en installer une sur une VM transfère une grande partie du travail d'exploitation au fournisseur. Le compromis, abordé dans FinOps et dépendance au fournisseur, est que plus un système utilise de services propres à un fournisseur, plus il est difficile à déplacer.

ATTENTION

Le modèle de responsabilité partagée

Passer au cloud ne transfère pas toute la responsabilité. Les fournisseurs l'indiquent clairement : ils sécurisent le cloud lui-même (bâtiments, matériel, couche de virtualisation) et le client sécurise ce qu'il y met : ses données, ses droits d'accès, sa configuration et, pour l'IaaS, le système d'exploitation. Beaucoup de fuites de données très médiatisées dans le cloud ne venaient pas d'une défaillance du fournisseur, mais d'un stockage mal configuré par le client et laissé ouvert sur internet.

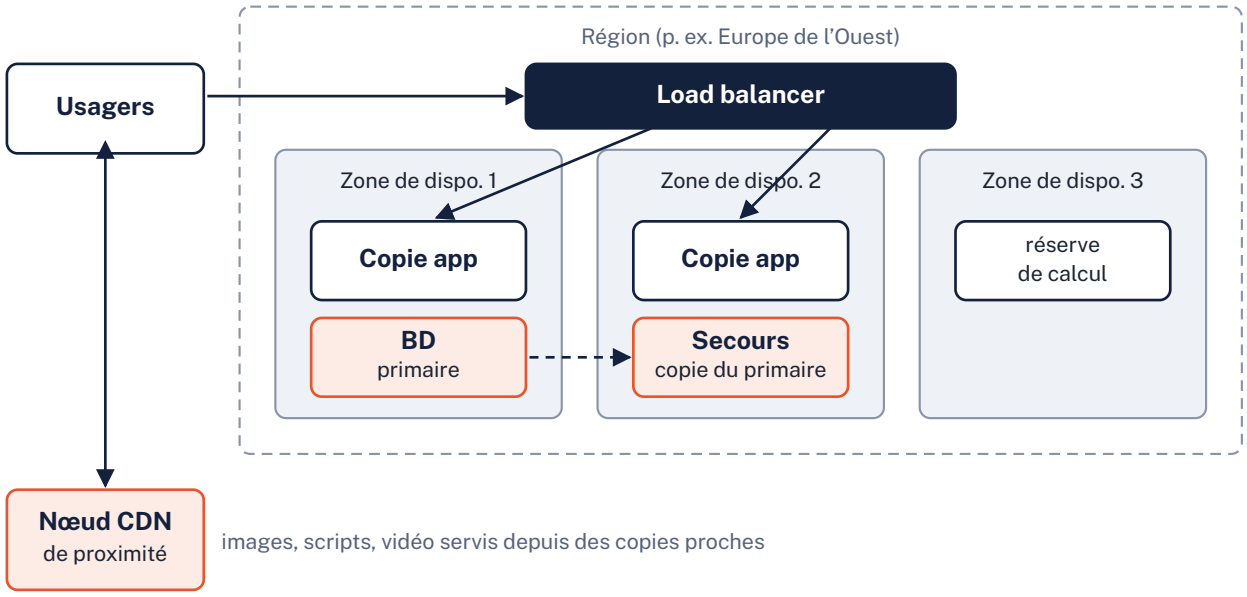
Régions, zones de disponibilité et réseaux de diffusion de contenu

Les fournisseurs de cloud regroupent leurs centres de données par zone géographique. Une région est une zone géographique (comme l'Irlande, Francfort ou la Virginie) dotée de son propre ensemble distinct de centres de données, isolé des autres régions. Chaque région est divisée en plusieurs zones de disponibilité : des groupes physiquement séparés d'un ou plusieurs centres de données, chacun avec son alimentation, son refroidissement et son réseau indépendants, généralement distants de plusieurs kilomètres mais assez proches pour être reliés par des connexions rapides. Un incendie, une inondation ou une panne de courant ne devrait mettre hors service qu'une zone au plus.

Cela offre aux architectes tout un éventail de protections. Faire tourner des copies d'une application dans deux ou trois zones d'une même région permet de survivre à la perte d'un centre de données pour un faible surcoût de latence ; les faire tourner dans deux régions permet de survivre à la perte d'une région entière, mais rend bien plus difficile la synchronisation des données. Les régions comptent aussi pour le droit et pour la latence : les règles de protection des données peuvent exiger que les données personnelles restent dans une juridiction donnée, et les utilisateurs sont servis plus vite depuis une région proche d'eux, car aucun réseau ne peut battre la vitesse de la lumière (voir comment les ordinateurs communiquent).

Un réseau de diffusion de contenu (CDN) va plus loin pour le contenu qui ne change pas d'un utilisateur à l'autre, comme les images, les feuilles de style, les scripts et la vidéo. Il en conserve des copies sur des serveurs situés dans des centaines d'endroits, appelés emplacements en périphérie (edge locations), de sorte qu'un utilisateur à Nairobi récupère les photos de produits de la boutique depuis une ville proche plutôt que depuis un centre de données en Europe. La page se charge plus vite, et les serveurs de la boutique traitent bien moins de requêtes. De nombreux CDN absorbent aussi le trafic d'attaque et peuvent exécuter de petits morceaux de code en périphérie.

À l'intérieur d'une région



Des copies de l'application dans deux zones de disponibilité maintiennent le service si une zone tombe en panne, tandis qu'un CDN sert le contenu statique depuis des emplacements proches des utilisateurs.

Scalabilité verticale et scalabilité horizontale

Quand un système doit traiter davantage de travail, il existe deux réponses de base. La scalabilité verticale (scaling up) consiste à prendre une machine plus grosse : plus de cœurs, plus de mémoire. Elle n'exige aucune modification du logiciel, ce qui est son grand avantage, mais on ne peut pas acheter plus gros que la plus grosse machine du marché, le prix grimpe de façon disproportionnée vers le haut de gamme, et une seule machine reste un point unique de défaillance. La scalabilité horizontale (scaling out) consiste à répartir le travail sur davantage de machines, généralement derrière un équilibreur de charge (load balancer) qui distribue les requêtes entre elles et cesse d'en envoyer à celles qui tombent en panne. Elle permet d'aller bien plus loin et de survivre aux pannes de machines, mais seulement si le logiciel est conçu

pour cela : par exemple, un serveur ne doit pas conserver la session d'un utilisateur uniquement dans sa propre mémoire, ni enregistrer les fichiers envoyés uniquement sur son propre disque, si la requête suivante de cet utilisateur risque d'aboutir sur un autre serveur.

Le cloud rend la scalabilité horizontale attrayante, car ajouter une machine se résume à un appel d'API, et l'*autoscaling* peut ajouter et retirer des copies automatiquement selon la charge. Les bases de données sont généralement la partie la plus difficile à faire passer à l'échelle horizontalement, c'est pourquoi le chapitre 7 consacre une section à la réplication et au partitionnement.

EXEMPLE COMMENTÉ

Exemple commenté : héberger un site de réservation de billets

Une petite chaîne de théâtres vend des billets en ligne. Le trafic est faible la majeure partie de la semaine, mais quand un spectacle populaire est mis en vente, dix fois plus de personnes arrivent en quelques minutes.

Une VM louée (IaaS)	Simple et bon marché. L'équipe applique les correctifs de l'OS. Le jour de la mise en vente, l'unique machine est submergée, et si elle tombe en panne, le site est hors service.
VM plus grosse	La scalabilité verticale donne de la marge sans modifier le code, mais le théâtre paie la grosse machine toute la semaine pour couvrir quelques heures de pointe.
App sur un PaaS, deux zones, autoscaling	Des copies de l'app tournent dans deux zones de disponibilité derrière un équilibreur de charge. Le jour de la mise en vente, la plateforme ajoute automatiquement des copies, puis les retire. Cela exige que l'app ne garde aucune donnée de session en mémoire locale.
Base de données managée	Le fournisseur fait tourner une base primaire dans une zone et une base de secours dans une autre, avec des sauvegardes automatiques. L'équipe conçoit toujours le schéma et les index (voir stocker des données).
CDN pour les images et les scripts	Les plans de salle et les photos des spectacles sont servis depuis des emplacements en périphérie, si bien que les serveurs de l'app consacrent leurs efforts aux réservations.

La conception finale coûte plus cher à mettre en place qu'une seule VM, mais moins cher à faire tourner qu'une machine surdimensionnée en permanence, et elle survit à la fois à un pic de trafic et à la perte d'un centre de données.

Chaque étape a échangé du contrôle contre de la commodité : l'équipe a choisi quel travail garder et lequel confier au fournisseur.

Pourquoi c’est important pour l’architecture

L’endroit où s’exécute un logiciel est désormais une décision d’architecture, et non un détail réglé après coup. Le choix entre VM, conteneurs, plateformes et fonctions façonne le coût, la rapidité avec laquelle l’équipe peut livrer, la manière dont le système passe à l’échelle et ce qui se passe quand un centre de données tombe en panne. Les régions et les zones transforment les objectifs de disponibilité en plans de déploiement concrets ; les services managés échangent du travail d’exploitation contre une dépendance à un fournisseur. Le chapitre 9 développe les stratégies de scalabilité, et le chapitre 10 traite des modèles de service et de déploiement cloud, des conteneurs et de l’infrastructure de manière approfondie, au niveau professionnel.

VÉRIFIEZ VOTRE COMPRÉHENSION

▼ Pourquoi les conteneurs démarrent-ils plus vite et occupent-ils moins d’espace que les machines virtuelles ?

Un conteneur ne démarre pas son propre système d’exploitation : c’est un processus isolé qui partage le noyau de l’hôte et n’embarque que l’application et ses bibliothèques. Une VM embarque et démarre un système d’exploitation invité complet.

▼ Une équipe loue une VM dans le cloud et y installe sa propre base de données. Qui est responsable des correctifs du système d’exploitation ?

L’équipe. Avec l’IaaS, le fournisseur gère le matériel et la virtualisation, et le client gère le système d’exploitation et tout ce qui se trouve au-dessus.

▼ Contre quelle défaillance protège le fait de tourner dans deux zones de disponibilité, et contre quoi ne protège-t-il pas ?

Il protège contre la perte d’un centre de données ou d’une zone (alimentation, incendie, réseau). Il ne protège pas contre la perte de la région entière, ni contre les bugs logiciels et les données erronées, qui touchent toutes les copies.

▼ Pourquoi une application qui fonctionne bien sur un seul serveur pourrait-elle mal se comporter une fois mise à l’échelle horizontalement derrière un équilibreur de charge ?

Elle peut conserver un état, comme les sessions des utilisateurs, dans sa propre mémoire : quand l’équilibreur de charge envoie la requête suivante à une autre copie, celle-ci ne connaît pas l’utilisateur.

▼ **À vous : Le site d'une chaîne de théâtres tourne sur une seule machine virtuelle louée. Le trafic est faible, mais un spectacle populaire amène dix fois plus de monde pendant une heure. Comparez l'achat d'une VM plus grosse et un PaaS sur deux zones avec autoscaling.**

Une VM plus grosse n'exige aucun changement de code, mais le théâtre paie la taille de pointe toute la semaine et une seule panne met encore le site hors service. Un PaaS sur deux zones avec autoscaling ne paie les copies supplémentaires qu'en cas de besoin et survit à la perte d'une zone, mais l'application ne doit conserver aucune donnée importante sur une seule copie. Pour des pics rares et brusques, l'option PaaS convient mieux. Le théâtre reste responsable de ses propres données et de sa configuration.

SOURCES

- NIST SP 800-145: The NIST Definition of Cloud Computing (2011): https://csrc.nist.gov/pubs/sp/800/145/final
- Docker: What is a container?: https://www.docker.com/resources/what-container/
- AWS documentation: Regions and Availability Zones: https://docs.aws.amazon.com/AWSEC2/latest/UserGuide/using-regions-availability-zones.html
- Microsoft Learn: What are availability zones?: https://learn.microsoft.com/en-us/azure/reliability/availability-zones-overview
- MDN Web Docs: CDN (glossary): https://developer.mozilla.org/en-US/docs/Glossary/CDN

SECTION 0.8 SUR 0.8 · DÉBUTANT · 21 MIN DE LECTURE · DERNIÈRE RELECTURE : NOVEMBRE 2026

Exigences, gestion de versions, tests, livraison et parcours d'une requête

L'IDÉE EN BREF

Construire un logiciel est un sport d'équipe qui suit une routine soignée. L'équipe consigne ce dont les utilisateurs ont besoin sous forme de courts récits, conserve chaque modification du code dans un historique partagé pour que rien ne se perde, fait vérifier chaque modification par un collègue et lance des tests automatiques qui servent de filet de sécurité. Les modifications passent ensuite par des copies d'entraînement du système avant d'atteindre la vraie, celle qu'utilisent les clients. À la fin de la section, on suit un simple appui sur le bouton « Commander » à travers chaque partie d'un système de livraison de repas.

CE QU'IL FAUT SAVOIR

- Les exigences décrivent ce qu'un système doit faire et avec quel niveau de qualité ; les récits utilisateur (En tant que..., je veux..., afin de...) assortis de critères d'acceptation sont une manière légère et courante de saisir les besoins fonctionnels.
- Git enregistre chaque modification sous forme de commits ; les branches isolent le travail en cours, et les pull requests apportent une revue de code et des vérifications automatisées avant une fusion dans main.
- La pyramide des tests recommande beaucoup de tests unitaires rapides, moins de tests d'intégration et peu de tests de bout en bout, qui sont lents.
- Les modifications passent par des environnements de développement, de test et de staging avant la production ; plus le staging ressemble à la production, plus il détecte de problèmes.
- L'intégration continue vérifie automatiquement chaque modification ; la livraison continue garde le logiciel toujours prêt à être mis en production ; le déploiement continu met automatiquement en production chaque modification qui passe les vérifications.
- Le cycle en cascade, Scrum et Kanban organisent le travail différemment ; le DevOps est une culture de responsabilité partagée pour construire et exploiter les logiciels.
- Une simple requête dans un système réel traverse le DNS, HTTPS, un équilibreur de charge, des services, une transaction de base de données, des API tierces et des files d'attente ; décider de la façon dont ces parties sont agencées, c'est l'architecture logicielle.

D’une idée à une exigence

Un logiciel commence par un besoin : un restaurant veut prendre des commandes en ligne, un hôpital veut que les patients prennent eux-mêmes leurs rendez-vous. Transformer ce besoin en quelque chose qu’une équipe peut construire, c’est le travail des exigences. Certaines exigences décrivent ce que le système doit faire (« un client peut payer par carte ») ; d’autres décrivent avec quel niveau de qualité il doit le faire (« le paiement se termine en moins de deux secondes pour 95 % des clients »). Le chapitre 2 montre que la seconde catégorie, les attributs de qualité, guide la plupart des décisions d’architecture.

Beaucoup d’équipes saisissent les besoins fonctionnels sous forme de récits utilisateur (user stories) : de courtes descriptions écrites du point de vue de l’utilisateur. Kent Beck les a introduits dans l’Extreme Programming à la fin des années 1990, et une équipe de l’entreprise londonienne Connextra a conçu en 2001 le modèle que la plupart des gens utilisent aujourd’hui :

```
As a hungry customer,
I want to reorder a meal I have ordered before,
so that I can eat sooner without searching the menu again.

Acceptance criteria
- My last 10 orders appear on the home screen.
- Tapping "Reorder" fills the basket with the same items.
- Items that are no longer available are flagged, not silently dropped.
```

Un récit utilisateur avec ses critères d’acceptation. Ces critères sont ce que les testeurs, et les tests automatisés, vérifieront.

Un récit est volontairement court et incomplet. Ron Jeffries le décrivait comme une carte, une conversation et une confirmation : la carte sert d’aide-mémoire, les détails se règlent en conversation entre les personnes qui veulent la fonctionnalité et celles qui la construisent, et les critères d’acceptation confirment quand elle est terminée. Les praticiens ne sont pas d’accord sur ce qu’il faut écrire à l’avance. Les équipes des secteurs réglementés, comme les dispositifs médicaux et l’aéronautique, doivent tenir des exigences formelles et traçables ; les équipes produit préfèrent souvent des récits plus légers, affinés juste avant le début du travail. Les deux approches sont légitimes dans leur contexte.

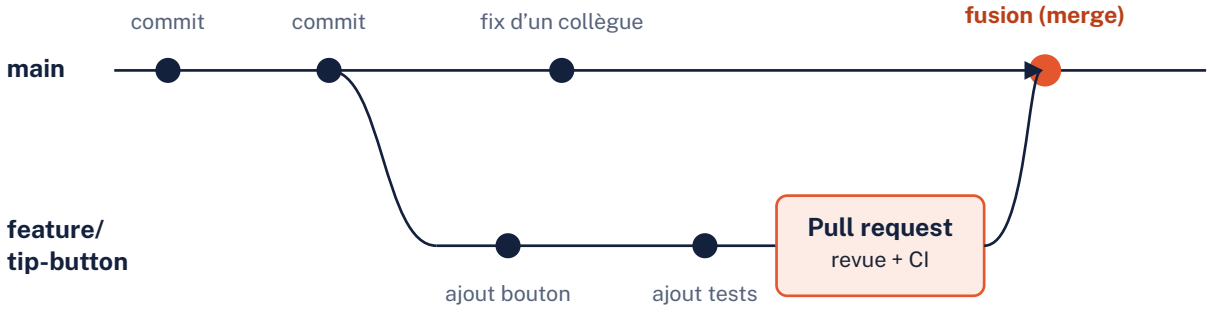
La gestion de versions avec Git

Le code source est le document partagé le plus précieux de l’équipe, et de nombreuses personnes le modifient chaque jour. La gestion de versions enregistre chaque modification, qui l’a faite et pourquoi, permet de travailler en parallèle sans écraser le travail des autres, et

permet de restaurer n'importe quel état antérieur. Presque toutes les équipes utilisent aujourd'hui Git, créé par Linus Torvalds en 2005 pour le développement du noyau Linux, généralement avec un service d'hébergement comme GitHub, GitLab ou Bitbucket.

Quelques notions couvrent l'essentiel de l'usage quotidien. Un commit est un instantané enregistré d'un ensemble de modifications, accompagné d'un message qui l'explique (« Ajouter un bouton de pourboire au paiement »). Une branche est une ligne de travail séparée : un développeur crée une branche à partir de `main`, y fait des commits sans déranger personne et, une fois terminé, *fusionne* (merge) sa branche dans la ligne principale, ce qui combine les deux historiques. Si deux personnes ont modifié les mêmes lignes, Git signale un *conflit de fusion* qu'un humain doit résoudre.

Sur les services d'hébergement, la fusion passe généralement par une pull request (GitLab parle de merge request) : une page web qui présente la modification proposée, où des vérifications automatisées s'exécutent et où des collègues effectuent une revue de code, en lisant la modification, en posant des questions et en suggérant des améliorations avant de l'approuver. La revue détecte des défauts, mais ses bénéfices les plus importants sont la diffusion des connaissances (au moins deux personnes comprennent chaque modification) et la cohérence de la conception. Les architectes participent souvent à la revue des modifications qui touchent la structure ou les interfaces.



La branche tient le travail en cours à l'écart de main jusqu'à sa revue et ses tests.

Branche, commit, revue, fusion : le cycle quotidien de la gestion de versions.

```
git switch -c feature/tip-button     # create a branch and move onto it
# ... edit files ...
git add checkout.py test_checkout.py
git commit -m "Add tip button to checkout"
git push -u origin feature/tip-button  # upload; then open a pull request
# after review and green checks, the branch is merged into main
```

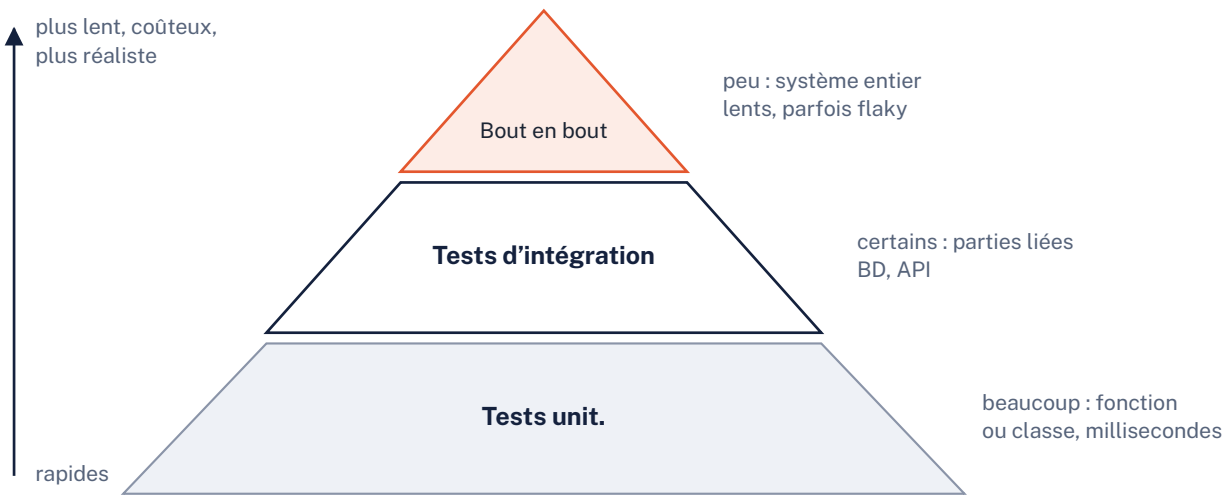
Le cycle Git quotidien en ligne de commande.

Les équipes divergent sur la gestion des branches. Certaines gardent des branches de longue durée par fonctionnalité ou par version ; d'autres pratiquent le *trunk-based development* (développement basé sur le tronc), en fusionnant de petites modifications dans `main` au moins une fois par jour et en masquant les fonctionnalités inachevées derrière des interrupteurs appelés feature flags. Les recherches du programme DORA associent les branches de courte durée à de meilleures performances de livraison, mais le bon choix dépend aussi de la manière dont l'équipe met en production.

Les tests : la pyramide

Toute modification risque de casser quelque chose qui fonctionnait, ce qu'on appelle une *régression*. Les tests automatisés sont des programmes qui vérifient d'autres programmes : on peut donc les lancer à chaque modification en quelques minutes, au lieu de confier la vérification à des personnes pendant des jours. Ils se présentent en couches. Un test unitaire vérifie un petit élément, comme une fonction, de façon isolée, et s'exécute en quelques millisecondes. Un test d'intégration vérifie que des parties fonctionnent ensemble : le code des commandes avec une vraie base de données, ou la boutique avec l'environnement de test d'un prestataire de paiement. Un test de bout en bout pilote le système entier comme le ferait un utilisateur, par exemple en parcourant l'app de la carte jusqu'à la confirmation de commande.

La pyramide des tests de Mike Cohn, publiée dans son livre de 2009 *Succeeding with Agile*, suggère les proportions : beaucoup de tests unitaires, moins de tests d'intégration et peu de tests de bout en bout. Les couches supérieures sont plus réalistes, mais plus lentes, plus coûteuses à maintenir et plus sujettes aux échecs *instables* (flaky), qui apparaissent et disparaissent pour des raisons sans rapport avec la modification. Certains praticiens défendent une autre forme, par exemple davantage de tests d'intégration pour les systèmes qui servent surtout de liant entre des services ; le principe sur lequel tous s'accordent est de détecter la plupart des problèmes avec des tests rapides, fiables et ciblés.



La forme indique des proportions, pas une loi : visez surtout des vérifications rapides et ciblées.

Beaucoup de tests unitaires rapides, moins de tests d'intégration et quelques tests de bout en bout donnent un retour rapide sans sacrifier le réalisme.

```
def delivery_fee(basket_pence, distance_km):   # money in whole pence
    if basket_pence ⩾ 2500:
        return 0                        # free delivery on larger orders
    return 150 + round(40 * distance_km)

def test_free_delivery_over_threshold():
    assert delivery_fee(3000, 5) == 0

def test_fee_grows_with_distance():
    assert delivery_fee(1000, 5) == 350
```

Deux tests unitaires écrits avec pytest pour une fonction de calcul des frais de livraison. Ils s'exécutent en quelques millisecondes et fixent les règles.

Les environnements : du développement à la production

Le code traverse plusieurs environnements sur le chemin qui le mène aux clients, chacun étant une copie séparée du système avec ses propres serveurs, données et réglages. Le but est de trouver les problèmes là où ils sont peu coûteux et sans danger, avant qu'ils n'atteignent la production, l'environnement réel dont dépendent les vrais utilisateurs.

Environnement	Qui l'utilise	Données	Objectif
Développement	Un développeur, sur un ordinateur portable ou un espace cloud personnel	Réduites, fictives	Écrire et essayer du code rapidement.
Test (ou intégration)	L'équipe et le pipeline automatisé	Fictives ou anonymisées	Lancer des tests automatisés et exploratoires sur les modifications combinées.
Staging (ou préproduction)	Testeurs, product owners, parfois des clients sélectionnés	Semblables à la production en forme et en volume, jamais de données personnelles brutes sans garanties	Répéter la mise en production sur une copie aussi proche que possible de la production.
Production	Vrais clients	Réelles	Faire tourner l'activité. Les modifications n'arrivent que par le pipeline.

Plus les environnements diffèrent de la production, plus les problèmes passent entre les mailles : une base de test de cent lignes ne révélera pas l'index manquant qui fait durer une requête quarante secondes face à dix millions de lignes. Il est plus facile de garder des environnements semblables quand ils sont construits à partir des mêmes images de conteneur et de la même configuration scriptée, un thème de l'infrastructure en tant que code.

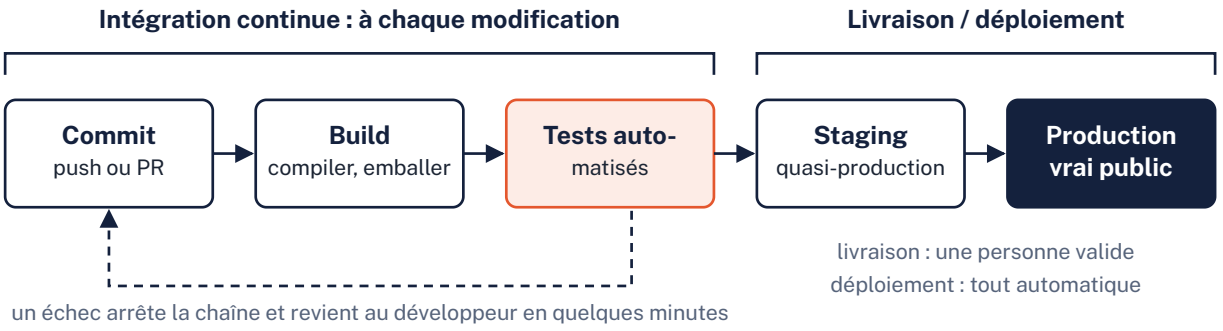
Intégration, livraison et déploiement continus

L'intégration continue (CI) est la pratique qui consiste à fusionner fréquemment, au moins une fois par jour, le travail de chacun dans la ligne principale partagée, un build et des tests automatisés vérifiant chaque modification. Elle est née de l'Extreme Programming dans les années 1990. Son intérêt est que les problèmes d'intégration sont détectés en quelques minutes, tant que la modification est fraîche dans l'esprit de son auteur, au lieu de l'être lors d'une pénible « phase d'intégration » des semaines plus tard. Une équipe qui pratique la CI traite un build en échec comme une urgence : le réparer passe avant tout nouveau travail.

La livraison continue (CD) prolonge la CI pour que le logiciel soit toujours prêt à être mis en production : chaque modification qui passe le pipeline pourrait être mise en production d'une simple pression sur un bouton, et la mise en production devient une décision métier de routine plutôt qu'un événement risqué. L'approche a été exposée par Jez Humble et David Farley dans

leur livre de 2010 *Continuous Delivery*. Le déploiement continu va un cran plus loin et met automatiquement en production chaque modification qui passe le pipeline, sans approbation humaine ; les équipes qui le pratiquent peuvent déployer de nombreuses fois par jour.

L'automatisation qui achemine une modification du commit jusqu'à la production est un pipeline. Comme chaque étape est automatisée et identique à chaque fois, les mises en production cessent de dépendre d'une personne qui doit se souvenir d'une liste de contrôle à minuit.



L'intégration continue vérifie automatiquement chaque modification ; la livraison et le déploiement continus acheminent le build validé jusqu'à la production.

Liste de contrôle : avant qu'une modification n'atteigne la production

Ce que le processus d'une équipe mature vérifie, le plus souvent automatiquement, pour chaque modification.

- [ ] **La modification est liée à un récit ou à un ticket**
  N'importe qui pourra retrouver plus tard pourquoi elle a été faite.
- [ ] **Elle est petite**
  Les petites modifications sont plus faciles à relire, à tester, à déployer et, si nécessaire, à annuler.
- [ ] **Un collègue l'a relue**
  Par une pull request, avec des questions qui ont obtenu réponse et des commentaires résolus.
- [ ] **Les tests automatisés passent à tous les niveaux**
  Tests unitaires et d'intégration dans la CI, et les vérifications de bout en bout qui couvrent le parcours concerné.
- [ ] **Elle a tourné en staging**
  Le même build, le même mécanisme de configuration et les mêmes étapes de déploiement qu'en production.
- [ ] **Il existe un chemin de retour**
  Un retour arrière (rollback) ou un feature flag peut la désactiver rapidement en cas de problème.
- [ ] **Quelqu'un remarquera si elle casse quelque chose**
  La supervision (monitoring) et les alertes couvrent ce que la modification touche (chapitre 9, observabilité).

Façons de travailler : cycle en cascade, Scrum et Kanban

Les équipes choisissent aussi comment organiser le travail lui-même. Dans une approche séquentielle, souvent appelée *cycle en cascade* (waterfall), le projet traverse des phases dans l'ordre : exigences, conception, implémentation, tests, déploiement. Chaque phase se termine avant que la suivante commence. Le nom est généralement attribué à l'article de Winston Royce de 1970, bien que Royce n'ait pas employé ce mot et qu'il ait en fait averti qu'un passage unique à travers les phases était risqué. Les plans séquentiels conviennent encore aux travaux dont les exigences sont figées et où le changement coûte très cher, comme certains projets

réglementés ou dépendants du matériel. Leur faiblesse, pour la plupart des logiciels, est que les utilisateurs ne voient rien avant un stade tardif : les malentendus apparaissent donc quand ils coûtent le plus cher à corriger.

Les approches agiles, nommées dans le Manifeste agile de 2001, livrent un logiciel qui fonctionne par cycles courts et ajustent les plans à mesure qu'elles apprennent. Scrum est le cadre le plus utilisé : une petite équipe travaille par cycles fixes appelés sprints, d'un mois ou moins, chacun commençant par une planification et se terminant par une revue du travail et une rétrospective sur la façon dont l'équipe a travaillé. Le Guide Scrum de 2020 définit trois responsabilités : le Product Owner (qui décide de ce qui a le plus de valeur), le Scrum Master (qui aide l'équipe à utiliser Scrum efficacement) et les Developers (qui font le travail). Kanban, adapté au logiciel à partir du système de production de Toyota, n'a pas de cycles fixes : les éléments de travail circulent sur un tableau visible (à faire, en cours, en revue, terminé), et l'équipe limite le nombre d'éléments en cours à un moment donné, afin que le travail soit terminé avant d'en commencer davantage.

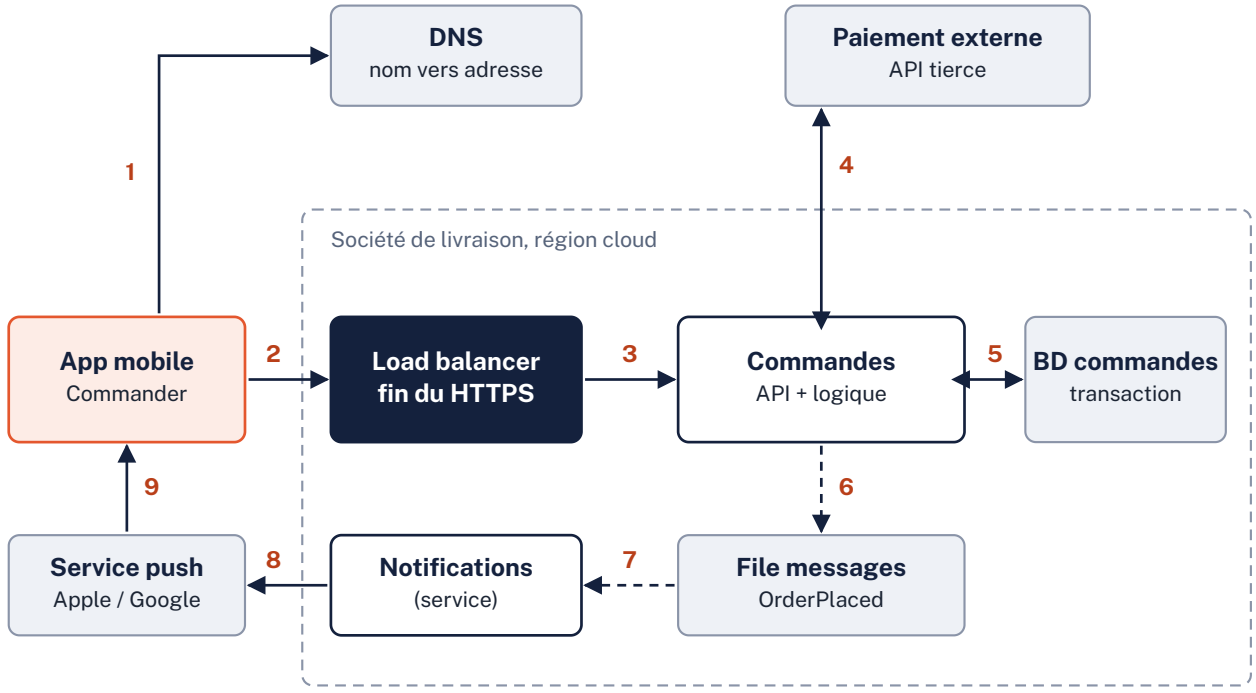
Aucune de ces approches ne convient à toutes les équipes, et beaucoup d'organisations les combinent : Scrum pour les équipes produit, Kanban pour le support et l'exploitation, un plan plus séquentiel autour d'une échéance réglementaire fixe. Qualifier un processus d'« agile » ne le rend pas agile ; ce qui compte, c'est qu'un logiciel qui fonctionne parvienne à de vrais utilisateurs assez souvent pour apprendre d'eux.

DevOps : une seule équipe responsable, du code jusqu'au client

Traditionnellement, les développeurs écrivaient le logiciel et une équipe d'exploitation distincte le faisait tourner, avec des incitations contradictoires : les développeurs étaient récompensés pour le changement, l'exploitation pour la stabilité. Le DevOps est le mouvement, baptisé lors de la première conférence DevOpsDays à Gand en 2009, qui réunit les deux. C'est avant tout une culture de responsabilité partagée (« you build it, you run it » : vous le construisez, vous l'exploitez), soutenue par des pratiques comme le CI/CD, l'infrastructure en tant que code, la supervision et les revues d'incidents sans recherche de coupable. Ce n'est ni un intitulé de poste ni un outil, même si l'on vend les deux sous ce nom. Le programme de recherche DORA est surtout connu pour mesurer la performance de livraison avec un petit ensemble d'indicateurs : la fréquence de déploiement d'une équipe, le temps qu'une modification met à atteindre la production, la fréquence des échecs des modifications, la rapidité du rétablissement du service après une défaillance et la fréquence à laquelle un déploiement nécessite une reprise non planifiée.

Exemple commenté : le parcours d'un simple appui

Tout ce chapitre se rejoint en un seul instant. Un client a rempli un panier dans une app de livraison de repas et appuie sur **Commander**. En une seconde ou deux, l'écran affiche « Commande confirmée », et vingt minutes plus tard le téléphone vibre : « Votre livreur est en route ». Voici ce qui se passe, étape par étape, avec la notion derrière chaque étape et l'endroit où elle a été présentée.



Un appui traverse le téléphone, internet, la région cloud de l'entreprise et deux prestataires externes ; les numéros correspondent aux étapes ci-dessous.

Pas à pas

1 **0. L'app construit la requête**

L'app est un logiciel installé sur le téléphone (section 0.1). Elle rassemble le panier à partir de ses structures de données en mémoire (section 0.2) et les transforme en corps JSON : restaurant, articles, adresse de livraison et une clé de commande unique générée sur le téléphone.

2 **1. DNS**

L'app a besoin de l'adresse de `api.fooddelivery.example`. Le téléphone interroge un résolveur DNS, qui renvoie une adresse IP, souvent depuis un cache (section 0.4).

3 **2. HTTPS**
L'app ouvre une connexion TCP et une session TLS, en vérifiant le certificat du serveur, puis envoie `POST /v1/orders` avec un jeton d'accès qui prouve quel client est connecté (sections 0.4 et 0.6).

4 **3. Équilibreur de charge**
La requête atteint un équilibreur de charge dans la région cloud de l'entreprise. Il déchiffre le HTTPS et transmet la requête à l'une des copies saines du service des commandes, qui tournent dans des conteneurs répartis sur plusieurs zones de disponibilité (section 0.7).

5 **4. API et paiement**
On accède au service des commandes par son API REST. Son code, organisé en modules (section 0.3), valide le panier, puis appelle l'API web du prestataire de paiement pour autoriser la carte, en attendant la réponse de façon synchrone avec un délai d'expiration (timeout), car le client ne doit pas lire « confirmée » tant que le paiement n'est pas approuvé (section 0.6). Il envoie la clé de commande comme clé d'idempotence, pour que, si l'appel fait l'objet d'une nouvelle tentative, la carte ne soit débitée qu'une seule fois.

6 **5. Transaction de base de données**
Dans une seule transaction ACID, le service insère la commande et ses articles et enregistre la référence du paiement. Soit tout est enregistré, soit rien ne l'est ; une fois validée, la transaction survit à une panne (section 0.5). Un index sur l'identifiant du client rendra son historique de commandes rapide à charger.

7 **6. File d'attente**
Le service dépose un message `OrderPlaced` dans une file de messages et répond immédiatement `201 Created`. L'app affiche « Commande confirmée ». Tout le reste se déroule de façon asynchrone (section 0.6).

8 **7. Les autres services réagissent**
Le service de la tablette du restaurant, le service d'affectation des livreurs et le service de notification lisent chacun le message quand ils sont prêts. Si l'un d'eux est brièvement indisponible, ses messages attendent dans la file.

9 **8. Notification push**
Le service de notification réagit à `OrderPlaced`, puis plus tard à un message similaire quand un livreur est affecté, en appelant le service push géré par Apple ou Google, une autre API tierce, qui remet la notification sur le téléphone du client.

10

9. Et derrière tout cela

Chaque morceau de code sur ce chemin a été écrit à partir d'un récit utilisateur, relu dans une pull request, testé à plusieurs niveaux et mis en production par un pipeline CI/CD, en passant par le staging (cette section).

EXEMPLE COMMENTÉ

Exemple commenté : ce que nous apprend le parcours

Posez la question « et si ? » à chaque étape, et les questions d'architecture apparaissent.

Et si l'appel de paiement expire ?	Le débit a-t-il eu lieu ou non ? La clé d'idempotence permet au service des commandes de redemander sans risque, sans débiter deux fois.
Et si l'enregistrement en base de données échoue après le paiement ?	Le paiement et la base de données sont des systèmes distincts : aucune transaction ACID unique ne couvre les deux. La conception doit prévoir un moyen de rembourser ou de réessayer, un problème que le chapitre 6 résout avec les sagas.
Et si le service de notification est en panne ?	Rien de visible ne casse : les commandes sont toujours prises et les messages attendent dans la file. C'est la raison pour laquelle cette étape est asynchrone.
Et si le trafic de l'heure du dîner triple ?	L'équilibreur de charge répartit les requêtes sur davantage de copies du service des commandes, ajoutées par l'autoscaling ; la base de données devient alors le goulet d'étranglement probable.
Et si une mise en production casse l'étape de paiement ?	Les tests du pipeline devraient le détecter ; sinon, la supervision déclenche une alerte et l'équipe revient à la version précédente.

Aucune de ces questions ne porte sur la manière d'écrire une fonction. Elles portent sur la façon dont les parties sont agencées et sur ce que chaque partie promet aux autres.

Pourquoi c'est important pour l'architecture

Le parcours d'un simple appui mobilise une douzaine de composants, trois organisations et des communications à la fois synchrones et asynchrones, et il a bien fallu le concevoir : quelqu'un a décidé que le paiement est synchrone et que les notifications ne le sont pas, que les commandes résident dans une base de données relationnelle, que le service tourne dans des conteneurs sur deux zones, que le prestataire de paiement est isolé derrière un seul module. Décider de l'agencement de ces parties, des qualités qu'atteint chaque agencement et de ce qu'il coûte, et consigner pourquoi, c'est l'architecture logicielle. La manière dont le logiciel est

construit compte aussi pour l'architecture : une architecture qu'on ne peut pas tester par morceaux, ni mettre en production sans coordonner dix équipes, ralentit chaque modification. Le chapitre 1 commence par ce qu'est l'architecture logicielle.

VÉRIFIEZ VOTRE COMPRÉHENSION

▼ **Quelle est la différence entre livraison continue et déploiement continu ?**

Avec la livraison continue, chaque modification validée peut être mise en production d'une simple pression sur un bouton, mais c'est une personne qui décide quand. Avec le déploiement continu, chaque modification validée est mise en production automatiquement.

▼ **Pourquoi la pyramide des tests place-t-elle les tests unitaires à la base ?**

Ils sont rapides, peu coûteux et fiables : une équipe peut donc en lancer des milliers à chaque modification et obtenir un retour en quelques minutes. Les tests de bout en bout, plus lents et plus fragiles, sont réservés aux quelques parcours qui en ont besoin.

▼ **Dans le parcours d'un simple appui, pourquoi l'appel de paiement est-il synchrone, mais la notification asynchrone ?**

L'app ne doit pas afficher « confirmée » tant que le paiement n'est pas approuvé : elle attend donc cette réponse. La notification peut être envoyée un peu plus tard : elle passe donc par une file, et sa défaillance ne peut pas bloquer la prise de commande.

▼ **Quels sont les deux principaux avantages d'une pull request, au-delà de la détection des bugs ?**

Elle diffuse les connaissances, car au moins une autre personne comprend chaque modification, et elle maintient la cohérence de la conception du code, car les relecteurs peuvent remettre en question les choix structurels.

▼ **À vous : Une cheffe de produit dit : « Les remboursements doivent être faciles pour les clients. » Transformez cela en un récit utilisateur (user story) avec deux critères d'acceptation pour l'app de livraison de repas, et indiquez quel type de test conviendrait à la règle sur le montant du remboursement.**

User story : en tant que client, je veux demander le remboursement d'un article manquant, afin de ne pas payer un repas que je n'ai pas reçu. Critères : la demande peut être faite depuis la page de la commande dans les 48 heures suivant la livraison ; le remboursement est égal au prix de l'article et ne dépasse jamais le montant payé. La règle sur le montant se prête à des tests unitaires rapides, car c'est un pur calcul, avec un test de bout en bout pour le parcours complet.

SOURCES

- Martin Fowler: Continuous Integration: https://martinfowler.com/articles/continuousIntegration.html
- Martin Fowler: Continuous Delivery: https://martinfowler.com/bliki/ContinuousDelivery.html
- Ham Vocke: The Practical Test Pyramid (martinfowler.com): https://martinfowler.com/articles/practical-test-pyramid.html
- The 2020 Scrum Guide: https://scrumguides.org/scrum-guide.html
- Pro Git (Scott Chacon and Ben Straub), free online book: https://git-scm.com/book/en/v2

Lectures complémentaires

- *Code: The Hidden Language of Computer Hardware and Software (2nd edition)*, Charles Petzold (2022), Microsoft Press. Construit un ordinateur à partir d'interrupteurs et de codes, par étapes faciles à suivre ; la meilleure explication non technique de la façon dont le matériel exécute le logiciel.
  https://www.microsoftpressstore.com/store/code-the-hidden-language-of-computer-hardware-and-software-9780137909100
- *Grokking Algorithms (2nd edition)*, Aditya Y. Bhargava (2024), Manning. Une introduction illustrée et en douceur aux structures de données, aux algorithmes et à la notation grand O, idéale après la section sur les données et les algorithmes.
  https://www.manning.com/books/grokking-algorithms-second-edition
- *The Python Tutorial*, Python Software Foundation, docs.python.org. L'introduction officielle et gratuite au langage utilisé dans les exemples de code de ce chapitre, y compris les fonctions, les modules et les classes.
  https://docs.python.org/3/tutorial/index.html
- *Learn web development*, MDN Web Docs contributors, Mozilla. Des guides gratuits et soigneusement tenus à jour sur le fonctionnement du web, HTTP, HTML et JavaScript, écrits pour les débutants.
  https://developer.mozilla.org/en-US/docs/Learn_web_development
- *Computer Networking: A Top-Down Approach (9th edition)*, James F. Kurose and Keith W. Ross (2025), Pearson. Le manuel de référence sur les réseaux, qui part du web pour descendre jusqu'aux paquets ; à lire de façon sélective une fois les bases acquises.
- *Pro Git (2nd edition)*, Scott Chacon and Ben Straub (2014), Apress (free online at git-scm.com). Le livre officiel et gratuit sur Git : commits, branches, fusions et collaboration, tels qu'ils sont présentés dans la dernière section.
  https://git-scm.com/book/en/v2
- *The NIST Definition of Cloud Computing (SP 800-145)*, Peter Mell and Timothy Grance (2011), National Institute of Standards and Technology. Une définition courte et très citée du cloud computing, de ses modèles de service et de ses modèles de déploiement.
  https://csrc.nist.gov/pubs/sp/800/145/final
- *Designing Data-Intensive Applications (2nd edition)*, Martin Kleppmann and Chris Riccomini (2026), O'Reilly Media. La meilleure étape suivante une fois ce chapitre bien assimilé : comment les bases de données, les réseaux et les systèmes distribués se comportent réellement, et pourquoi c'est important.

Glossaire

ACID
Atomicité, cohérence, isolation, durabilité : les garanties qu'offre une transaction de base de données pour que les données restent correctes malgré les pannes et les utilisateurs simultanés.

Adresse IP
Adresse numérique qui identifie un appareil sur un réseau afin que des paquets puissent lui être remis, comme 192.0.2.10 (IPv4) ou 2001:db8::1 (IPv6).

Agile
Famille d'approches du développement logiciel, définies dans le Manifeste agile de 2001, qui livrent du logiciel fonctionnel par cycles courts et adaptent les plans en fonction des retours.

Algorithme
Méthode précise, étape par étape, pour résoudre un problème, comme trier une liste ou trouver l'itinéraire le plus court, et qui se termine toujours.

API (interface de programmation d'application)
Ensemble défini d'opérations, d'entrées et de sorties par lequel un logiciel en utilise un autre, sans avoir besoin de savoir comment il fonctionne à l'intérieur.

API web
API proposée sur un réseau, généralement au moyen de HTTP, afin que des programmes s'exécutant sur des machines différentes, écrits dans des langages différents, puissent fonctionner ensemble.

Appel synchrone
Requête pour laquelle l'appelant attend la réponse avant de continuer, comme dans un appel d'API HTTP ordinaire.

Bande passante
Quantité de données qu'une connexion peut transporter par seconde, par exemple 100 mégabits par seconde. Une bande passante élevée ne réduit pas la latence.

Base de données
Collection organisée de données, stockée de façon à pouvoir être interrogée, mise à jour et maintenue cohérente, généralement gérée par un système de gestion de base de données.

Base de données clé-valeur
Base de données qui stocke des valeurs sous des clés uniques et ne les retrouve que par leur clé, comme un dictionnaire très grand et très rapide.

Base de données managée
Service de base de données exploité par un fournisseur de cloud, qui prend en charge l'installation, les correctifs, les sauvegardes et le failover, tandis que le client gère le schéma, les requêtes et les accès.

Base de données orientée documents
Base de données NoSQL qui stocke chaque enregistrement sous forme de document autonome, généralement de type JSON, pouvant contenir des données imbriquées sans devoir partager un schéma fixe.

Base de données orientée graphe
Base de données qui stocke les éléments sous forme de nœuds et leurs relations sous forme d'arêtes, de sorte que le parcours de chaînes de connexions est rapide.

Base de données relationnelle
Base de données qui stocke les données dans des tables de lignes et de colonnes et relie les tables par des clés. On l'interroge en SQL.

Base orientée colonnes larges (wide-column store)
Base de données NoSQL qui regroupe les données en lignes identifiées par une clé, chacune comportant de nombreuses colonnes, réparties sur de nombreuses machines. Conçue pour de très grands volumes d'écriture.

Bibliothèque
Code réutilisable écrit par quelqu'un d'autre, que votre code appelle quand il en a besoin, par exemple une bibliothèque pour les dates ou pour l'envoi d'e-mails.

Bit
Chiffre binaire, 0 ou 1 : la plus petite unité d'information qu'un ordinateur stocke.

Branche
Ligne de développement distincte dans la gestion de versions, utilisée pour travailler sur une modification sans affecter la ligne principale jusqu'à ce qu'elle soit prête à être fusionnée.

Bug
Erreur dans un logiciel qui le fait se comporter autrement que prévu, d'un plantage à une réponse silencieusement fausse.

Bytecode
Instructions compactes destinées à une machine virtuelle plutôt qu'à un vrai CPU. Java et Python compilent le code source en bytecode, que leurs environnements d'exécution exécutent ensuite sur toute machine prise en charge.

Cache
Copie de données conservée à un endroit plus rapide d'accès, afin que les requêtes répétées évitent un travail lent. Elle échange de la mémoire supplémentaire, et le risque de données périmées, contre de la vitesse.

Centre de données
Bâtiment conçu pour héberger de nombreux serveurs avec une alimentation électrique fiable, un refroidissement, une sécurité physique et des connexions réseau.

Changement cassant (breaking change)
Modification d'une API qui fait cesser de fonctionner du code client existant et correct, comme le fait de renommer ou de supprimer un champ.

Classe et objet
Une classe est un modèle qui regroupe des données et les fonctions (méthodes) qui opèrent sur elles ; un objet est une instance créée à partir de ce modèle, comme le compte d'un client donné.

Clé d'API
Longue chaîne secrète qui identifie l'application appelante auprès d'une API. Quiconque détient la clé peut agir en tant que cette application ; elle doit donc rester secrète.

Clé étrangère
Colonne d'une table qui contient la clé primaire d'une ligne d'une autre table, ce qui relie les deux. La base de données peut refuser les valeurs qui ne renvoient à rien.

Clé primaire
Colonne, ou ensemble de colonnes, dont la valeur identifie de manière unique chaque ligne d'une table, comme customer_id.

Cloud computing (informatique en nuage)
Location de ressources informatiques (serveurs, stockage, bases de données et autres services) par internet, à la demande, en payant ce que l'on utilise.

Code d'état HTTP
Nombre à trois chiffres présent dans chaque réponse HTTP, qui indique comment s'est déroulée la requête : 2xx succès, 3xx redirection, 4xx problème dans la requête, 5xx problème du côté du serveur.

Code machine
Instructions codées sous forme de nombres qu'une famille donnée de CPU peut exécuter directement. Le code machine d'une famille (par exemple x86-64) ne s'exécute pas sur une autre (par exemple Arm).

Code source
Le texte lisible par l'humain d'un programme, écrit dans un langage de programmation comme Python ou Java, qui est ensuite compilé ou interprété pour qu'un ordinateur puisse l'exécuter.

Cohésion
Degré auquel le contenu d'une partie d'un système forme un tout. Une forte cohésion signifie qu'un module accomplit une seule tâche claire.

Commit
Instantané enregistré de modifications dans la gestion de versions, avec un auteur, un horodatage et un message expliquant la modification.

Compilateur
Programme qui traduit le code source sous une autre forme, comme du code machine ou du bytecode, avant l'exécution du programme.

Conteneur
Application empaquetée avec ses bibliothèques et ses paramètres, exécutée comme un processus isolé qui partage le noyau du système d'exploitation de l'hôte. Plus légère et plus rapide à démarrer qu'une machine virtuelle.

Contrat d'API
Promesse documentée que fait une API : quelles requêtes elle accepte, ce que chacune renvoie, quelles erreurs peuvent survenir et ce qui ne changera pas sans préavis.

Couplage
Degré auquel une partie d'un système dépend des détails d'une autre. Un couplage faible signifie qu'une modification dans une partie impose rarement des modifications ailleurs.

CPU (processeur)
Unité centrale de traitement : la puce qui exécute les instructions d'un programme, une étape simple à la fois, des milliards de fois par seconde. Les CPU modernes comportent plusieurs cœurs qui travaillent en parallèle.

Dépendance
Tout logiciel dont votre code a besoin pour fonctionner, comme une bibliothèque, un framework ou un autre service. S'il change ou tombe en panne, votre code peut tomber en panne lui aussi.

Déploiement continu
Mise en production automatique de chaque modification qui passe le pipeline, sans étape d'approbation manuelle.

DevOps
Culture et ensemble de pratiques dans lesquels le développement et l'exploitation travaillent comme une seule équipe, en partageant la responsabilité de la construction, de la mise en production et de l'exploitation du logiciel, avec l'appui de l'automatisation et de la mesure.

Dictionnaire (map)
Structure de données qui stocke des valeurs sous des clés uniques, de sorte qu'une valeur puisse être trouvée directement par sa clé, généralement en un temps à peu près constant. Aussi appelée map ou table de hachage.

DNS
Domain Name System : le service d'annuaire d'internet, qui transforme des noms comme www.example.com en adresses IP.

Docker
Ensemble d'outils très répandu, publié en 2013, pour construire des images de conteneur et exécuter des conteneurs. Son format d'image est devenu un standard ouvert.

Encapsulation
Fait de garder privés les données internes et les détails d'un objet ou d'un module, afin que le reste du code ne puisse l'utiliser qu'au moyen des opérations qu'il choisit d'offrir.

Environnement
Copie distincte et complète du système et de ses paramètres, utilisée dans un but précis, comme le développement, les tests, le staging ou la production.

Équilibreur de charge (load balancer)
Composant qui reçoit les requêtes entrantes et les répartit entre plusieurs serveurs, en ignorant ceux qui ne sont pas en bonne santé.

Exigence
Énoncé de ce qu'un système doit faire, ou d'une qualité qu'il doit avoir, convenu avec les personnes qui en ont besoin.

Fichier
Suite nommée d'octets stockée sur un disque et gérée par le système d'exploitation, comme une photo, une feuille de calcul ou un journal.

File de messages
Logiciel qui stocke les messages des expéditeurs jusqu'à ce que les destinataires soient prêts à les traiter, de sorte que les deux n'ont pas besoin de tourner en même temps ni d'aller à la même vitesse.

Fonction
Morceau de code nommé et réutilisable qui prend des entrées (paramètres), effectue un travail et renvoie généralement une sortie.

Framework
Code réutilisable qui fournit la structure d'ensemble d'une application et appelle votre code aux bons moments, comme Django ou Spring.

Gestion de versions
Système qui enregistre chaque modification d'un ensemble de fichiers, qui l'a faite et pourquoi, afin que les personnes puissent travailler en parallèle, relire les modifications et revenir à n'importe quelle version antérieure.

Gestionnaire de paquets
Outil comme pip (Python) ou npm (JavaScript) qui télécharge des bibliothèques et les bibliothèques dont elles dépendent, dans les versions qu'un projet spécifie.

Git
Système de gestion de versions le plus utilisé, créé par Linus Torvalds en 2005. Chaque développeur dispose d'une copie complète de l'historique du projet.

HTTP
Hypertext Transfer Protocol : le langage de requêtes et de réponses du web, dans lequel un client envoie une méthode comme GET à une URL et un serveur répond par un code d'état et un contenu.

Hyperviseur
Logiciel qui crée et exécute des machines virtuelles, en leur partageant les processeurs, la mémoire et les périphériques d'un ordinateur physique tout en les maintenant isolées.

IaaS (infrastructure en tant que service)
Modèle de cloud dans lequel vous louez des machines virtuelles, du stockage et des réseaux, et gérez vous-même le système d'exploitation et tout ce qui se trouve au-dessus.

Image de conteneur
Paquet versionné en lecture seule contenant une application et tout ce dont elle a besoin pour s'exécuter, à partir duquel des conteneurs identiques peuvent être démarrés n'importe où.

Index
Structure de données supplémentaire, conservée à côté des données, qui permet à un système de trouver des éléments d'après un champ choisi sans tout parcourir, comme l'index à la fin d'un livre.

Index de base de données
Structure de données supplémentaire, généralement un arbre B, qui permet à la base de données de trouver des lignes d'après la valeur d'une colonne sans lire toute la table. Il accélère les lectures mais ralentit les écritures et occupe de l'espace.

Intégration continue (CI)
Pratique consistant à fusionner fréquemment, au moins une fois par jour, les modifications de chaque développeur dans la ligne principale partagée, avec un build et des tests automatisés qui vérifient chaque modification.

Interface
Ensemble convenu d'opérations qu'un logiciel offre aux autres, avec leurs entrées et leurs sorties : un contrat qui cache la manière dont le travail est fait.

Interpréteur
Programme qui lit du code et exécute ses instructions au fur et à mesure, plutôt que de traduire tout le programme à l'avance.

Inversion de contrôle
Organisation dans laquelle un framework, plutôt que votre propre code, contrôle le déroulement du programme et appelle votre code lorsque c'est nécessaire.

Jeton d'accès (access token)
Justificatif de courte durée, délivré après la connexion d'un utilisateur ou d'une application, envoyé avec chaque requête d'API pour prouver ce que l'appelant a le droit de faire.

Jointure (join)
Opération SQL qui combine les lignes de deux tables ou plus en faisant correspondre des colonnes liées, par exemple les commandes avec les clients qui les ont passées.

JSON (JavaScript Object Notation)
Format texte pour données structurées composé d'objets (paires nom-valeur entre accolades), de tableaux (listes entre crochets), de chaînes, de nombres, de true, false et null. Normalisé par la RFC 8259.

Kanban
Manière de gérer le travail comme un flux continu sur un tableau visuel, en limitant le nombre d'éléments pouvant être en cours en même temps.

Latence
Le délai avant que des données arrivent ou qu'une réponse revienne, généralement mesuré en millisecondes. La distance, les sauts réseau et un traitement lent l'augmentent tous.

Limite de débit (rate limit)
Plafond du nombre de requêtes qu'un client peut faire sur une période, par exemple 100 par minute. Les requêtes au-delà sont refusées, généralement avec le code d'état HTTP 429.

Linux
Noyau de système d'exploitation libre et open source, utilisé avec d'autres logiciels comme système d'exploitation de la plupart des serveurs, des machines du cloud et des conteneurs, ainsi que des téléphones Android.

Livraison continue
Maintien du logiciel dans un état toujours livrable, de sorte que toute version qui passe le pipeline automatisé puisse être mise en production en appuyant sur un bouton.

Logiciel (software)
Les programmes, avec les données et la configuration dont ils ont besoin ; les instructions qui indiquent au matériel quoi faire.

Machine virtuelle (VM)
Imitation logicielle d'un ordinateur complet, avec son propre système d'exploitation, qui s'exécute à côté d'autres VM sur une même machine physique sous le contrôle d'un hyperviseur.

Matériel (hardware)
Parties physiques d'un ordinateur que l'on pourrait toucher : le processeur, la mémoire, le stockage, la carte réseau, l'écran, etc.

Messagerie asynchrone
Communication dans laquelle l'émetteur dépose un message quelque part pour que le destinataire le traite plus tard, et poursuit son travail sans attendre de réponse.

Modèle client-serveur
Organisation dans laquelle des clients (comme des navigateurs ou des apps mobiles) envoient des requêtes et des serveurs, qui détiennent les données et la logique partagées, y répondent.

Module
Unité de code, souvent un seul fichier, qui regroupe sous un même nom des fonctions et des données liées et décide de ce qu'elle met à disposition du reste du code. Un paquet (package) est un groupe de modules.

NoSQL
Terme générique désignant les bases de données qui n'utilisent pas le modèle relationnel en tables, comme les bases orientées documents, clé-valeur, colonnes larges et graphes.

Notation grand O (Big-O)
Manière de décrire comment le travail effectué par un algorithme augmente lorsque la quantité de données augmente, par exemple O(n) pour une croissance linéaire ou O(1) pour un temps constant.

Octet (byte)
Groupe de huit bits, pouvant prendre 256 valeurs différentes. La taille des fichiers et de la mémoire se compte en octets (kilo-octets, mégaoctets, gigaoctets).

PaaS (plateforme en tant que service)
Modèle de cloud dans lequel vous fournissez le code de votre application ou votre conteneur, et le fournisseur exécute, corrige et met à l'échelle la plateforme sous-jacente.

Paiement à l'usage (pay-as-you-go)
Modèle de tarification dans lequel les ressources ne vous sont facturées que pendant que vous les utilisez, par exemple à la seconde de temps serveur ou au gigaoctet stocké.

Paquet (réseau)
Petit bloc de données envoyé sur un réseau avec des informations d'adressage. Les messages volumineux sont découpés en de nombreux paquets, réassemblés à l'autre bout.

Pipeline CI/CD
Suite automatisée d'étapes (build, test, déploiement en staging, déploiement en production) par laquelle passe chaque modification pour parvenir aux utilisateurs.

Port
Nombre de 0 à 65535 qui identifie le programme d'un appareil qui doit recevoir le trafic réseau, par exemple 443 pour HTTPS.

Processus
Instance en cours d'exécution d'un programme, avec sa propre mémoire privée et sa part de temps CPU, créée et gérée par le système d'exploitation.

Production
L'environnement réel dont dépendent les vrais utilisateurs et les vraies données.

Programme
Liste d'instructions précises qu'un ordinateur suit pour accomplir une tâche. Un programme en cours d'exécution s'appelle un processus.

Pull request
Demande de fusion d'une branche dans la ligne principale, sur un service comme GitHub, où des collègues relisent le changement et où des vérifications automatisées s'exécutent avant qu'il soit accepté.

Pyramide des tests
Repère, dû à Mike Cohn, sur les proportions des tests automatisés : beaucoup de tests unitaires rapides, moins de tests d'intégration et seulement quelques tests de bout en bout, lents.

RAM (mémoire)
Mémoire vive (random-access memory) : stockage rapide et temporaire qui contient les programmes et les données utilisés en ce moment. Son contenu est perdu lorsque l'alimentation est coupée ou que le programme se termine.

Récit utilisateur (user story)
Courte description d'une fonctionnalité du point de vue de la personne qui la souhaite, souvent rédigée ainsi : En tant que [rôle], je veux [capacité], afin de [bénéfice].

Région
Zone géographique, comme l'Europe de l'Ouest ou l'Est des États-Unis, où un fournisseur de cloud exploite un groupe distinct de centres de données. Les régions sont isolées les unes des autres.

Réseau de diffusion de contenu (CDN)
Réseau de serveurs répartis en de nombreux endroits qui conserve des copies de contenus comme les images, les scripts et les vidéos à proximité des utilisateurs, afin qu'ils se chargent plus vite et que le serveur d'origine travaille moins.

REST (Representational State Transfer)
Style architectural pour les API web, décrit par Roy Fielding en 2000, dans lequel les clients agissent sur des ressources identifiées par des URL au moyen des méthodes HTTP standard.

Revue de code
Examen d'une modification de code proposée par des collègues avant sa fusion, afin de trouver des défauts, de partager les connaissances et de garder une conception cohérente.

SaaS (logiciel en tant que service)
Logiciel complet utilisé par Internet, généralement au moyen d'un navigateur ou d'une application, le fournisseur se chargeant de tout faire fonctionner.

Sauvegarde
Copie des données conservée séparément de l'original afin de pouvoir les restaurer après une perte, une corruption ou une suppression.

Scalabilité horizontale (scaling out)
Absorber davantage de charge en ajoutant des machines et en répartissant le travail entre elles, généralement derrière un équilibreur de charge.

Scalabilité verticale (scaling up)
Absorber davantage de charge en passant à une machine plus puissante, dotée de plus de processeurs ou de mémoire.

Schéma
La structure déclarée d'une base de données : ses tables, ses colonnes, ses types de données, ses clés et ses règles.

Scrum
Framework agile dans lequel une petite équipe travaille par cycles fixes appelés sprints, d'un mois ou moins, avec des événements, des responsabilités et des artefacts définis.

SDK (kit de développement logiciel)
Bibliothèque, généralement fournie par le propriétaire d'une API, qui enveloppe les appels HTTP de l'API dans des fonctions ordinaires pour un langage de programmation donné.

Serverless et FaaS (fonction en tant que service)
Modèle de cloud dans lequel vous envoyez des fonctions individuelles qui ne s'exécutent que lorsqu'un événement les déclenche ; le fournisseur gère tous les serveurs et vous payez à l'exécution.

Serveur
Ordinateur, ou programme qui s'exécute sur un ordinateur, qui attend les requêtes d'autres ordinateurs (les clients) et y répond.

Service managé
Service cloud, comme une base de données, une file de messages ou un service de stockage, que le fournisseur installe, exploite, corrige et met à l'échelle, afin que les clients l'utilisent sans avoir à le faire tourner eux-mêmes.

SQL (Structured Query Language)
Le langage standard pour définir, interroger et modifier les données des bases de données relationnelles. Vous décrivez le résultat voulu et la base de données détermine comment l'obtenir.

Structure de données

Manière d'organiser des données en mémoire, comme une liste, un dictionnaire, un ensemble, une file, une pile, un arbre ou un graphe, choisie pour rendre rapides les opérations dont vous avez besoin.

Système d'exploitation

Le logiciel qui gère le matériel d'un ordinateur et le partage entre les programmes : temps CPU, mémoire, fichiers, réseau et sécurité. Exemples : Linux, Windows, macOS, Android et iOS.

Système de gestion de base de données (SGBD, DBMS)

Logiciel qui stocke une base de données et en contrôle l'accès : il répond aux requêtes, fait respecter les règles, gère de nombreux utilisateurs à la fois et se rétablit après une panne. PostgreSQL et MySQL en sont des exemples.

TCP

Transmission Control Protocol : fournit un flux de données fiable et ordonné entre deux programmes, en renvoyant les paquets perdus. La plus grande partie du trafic web l'a traditionnellement utilisé.

Test d'intégration

Test automatisé qui vérifie que plusieurs parties fonctionnent correctement ensemble, comme du code avec une vraie base de données ou l'API d'un autre service.

Test de bout en bout

Test automatisé qui pilote l'ensemble du système comme le ferait un utilisateur, par exemple en parcourant une app à coups de touchers, afin de vérifier un parcours complet.

Test unitaire

Test automatisé qui vérifie isolément un petit morceau de code, comme une fonction, et qui s'exécute en quelques millisecondes.

Thread (fil d'exécution)

Une séquence d'instructions en cours d'exécution à l'intérieur d'un processus. Les threads d'un même processus partagent sa mémoire ; ils peuvent donc travailler en même temps, mais doivent se coordonner.

TLS et HTTPS

Transport Layer Security chiffre les données échangées entre deux programmes et, au moyen de certificats, prouve l'identité du serveur. HTTPS est du HTTP transporté sur TLS.

Transaction

Groupe d'opérations de base de données traité comme une seule unité : soit toutes prennent effet (commit), soit aucune (rollback).

Type de données

Nature de la valeur que représente une donnée, comme un entier, un nombre décimal, du texte ou vrai/faux, qui détermine les opérations qui ont un sens sur elle.

UDP

User Datagram Protocol : envoie des paquets individuels sans garantie de livraison ni d'ordre. Plus rapide et plus simple que TCP ; utilisé pour les appels vidéo, les jeux et le DNS.

UTF-8

La manière dominante d'encoder du texte en octets. Il peut représenter tous les caractères Unicode, avec un octet pour les lettres de base de l'anglais et jusqu'à quatre octets pour les autres.

Versionnage d'API

Publication des modifications d'une API sous un nouvel identifiant de version, comme /v2/, en maintenant l'ancienne version en service pendant un certain temps pour que les clients puissent migrer à leur rythme.

Webhook

Requête HTTP qu'un fournisseur envoie à une URL enregistrée par son client lorsqu'un événement se produit, comme un paiement réussi. Elle inverse le sens habituel d'un appel d'API.

Zone de disponibilité

L'un des groupes de centres de données physiquement séparés au sein d'une région cloud, chacun disposant de sa propre alimentation électrique, de son refroidissement et de son réseau, suffisamment proches pour des connexions rapides.