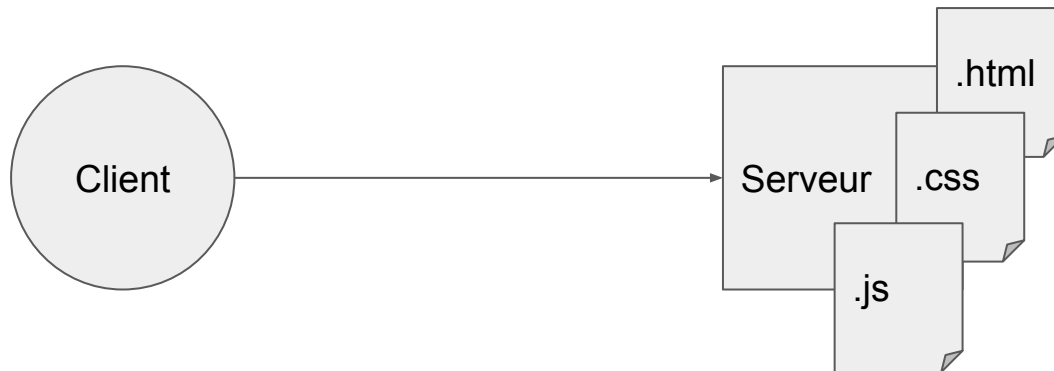


Révision

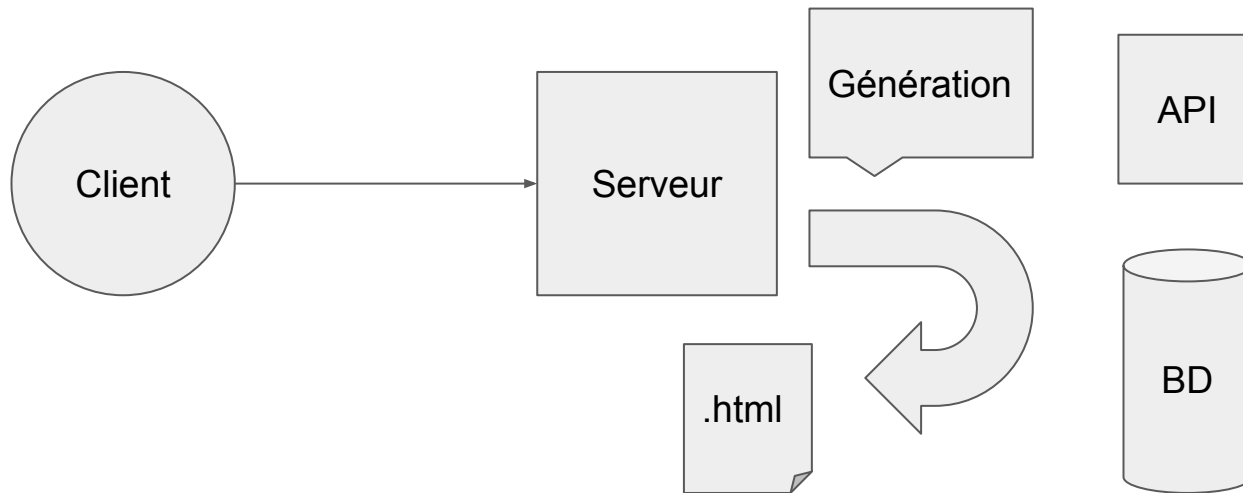
Sites statiques

Un site statique sert toujours les mêmes fichiers (HTML, CSS, js).



Sites dynamiques

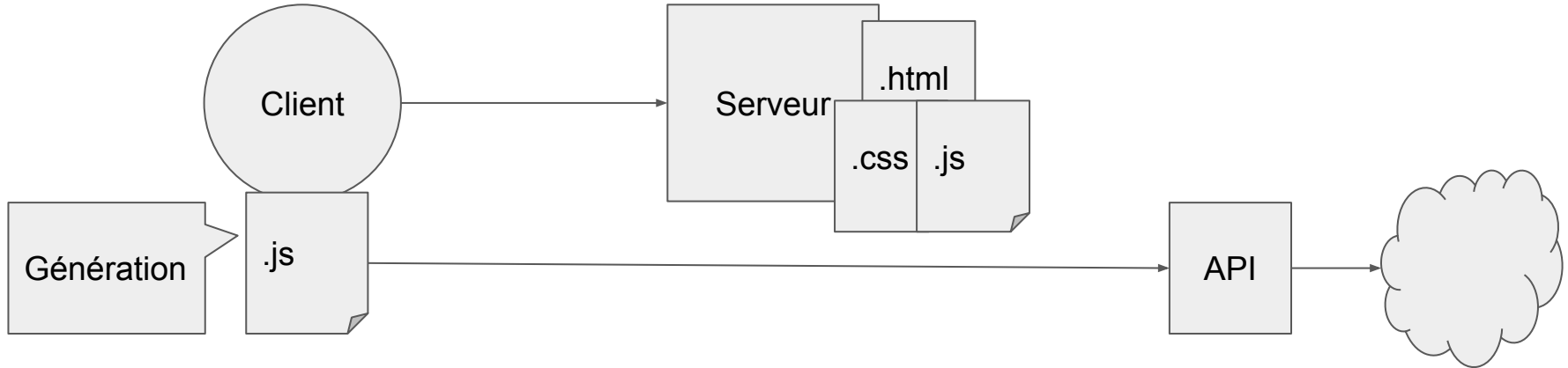
Le **serveur** génère les pages web dynamiquement en réponse aux interactions de l'utilisateur.



Application Monopage - SPA

Le client télécharge un ensemble de fichiers statiques depuis le serveur.

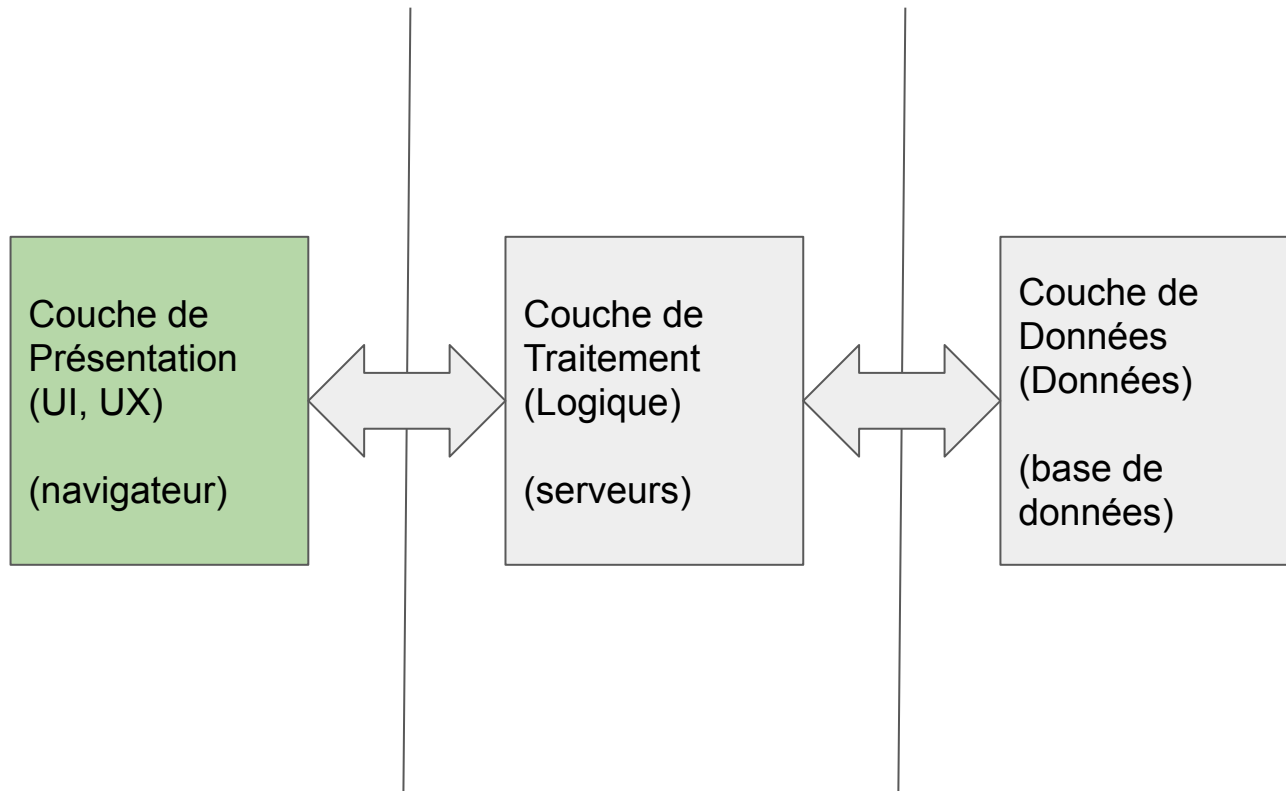
Le code javascript modifie l'apparence de la page dynamiquement et effectue des appels aux APIs pour charger (et modifier) les données.



Architecture en 3 couches (3-tier architecture)


L'architecture logique du système est divisée en trois couches

- couche de *présentation* - *UI, navigateur (e.g. js, Angular, bootstrap)*
- couche de *traitement* - *logique, serveur (e.g. php, spring, flask)*
- couche de *données* - *état persistant, base de données (e.g. mysql, mongo db)*



Composants en développement frontend

Pour le développement frontend, les composants servent à définir des éléments UI et combinent le HTML, CSS et javascript nécessaires à cet élément.


 Material


Components

CDK

Guides

14.2.1



 GitHub

Components

Autocomplete

Badge

Bottom Sheet

Button

Button toggle

Card

Checkbox

Chips

Core

Angular Material offers a wide variety of UI components based on the [Material Design specification](#)

Greetings

hello

hello world

Autocomplete
Suggests relevant options as the user types.

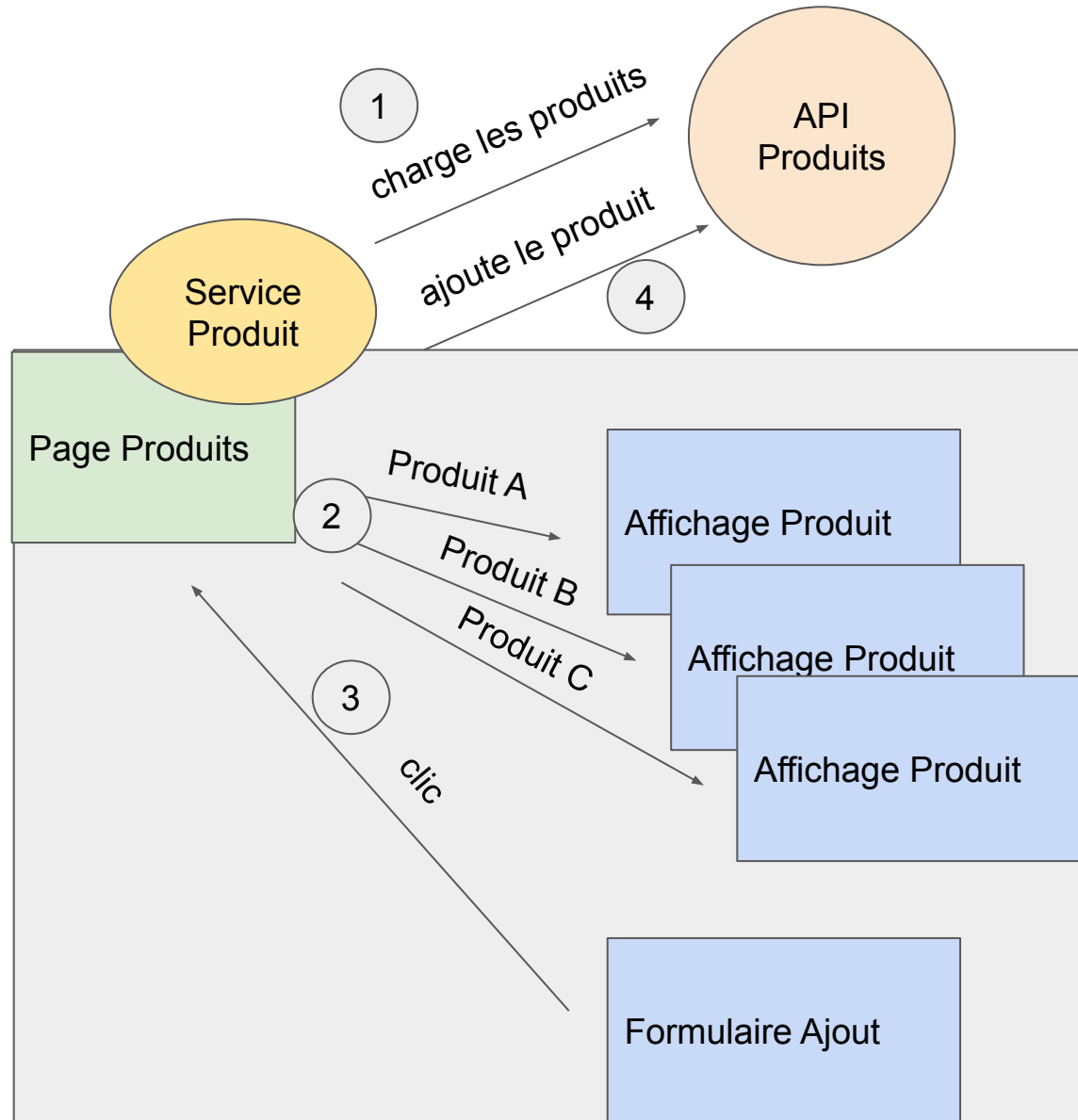
1

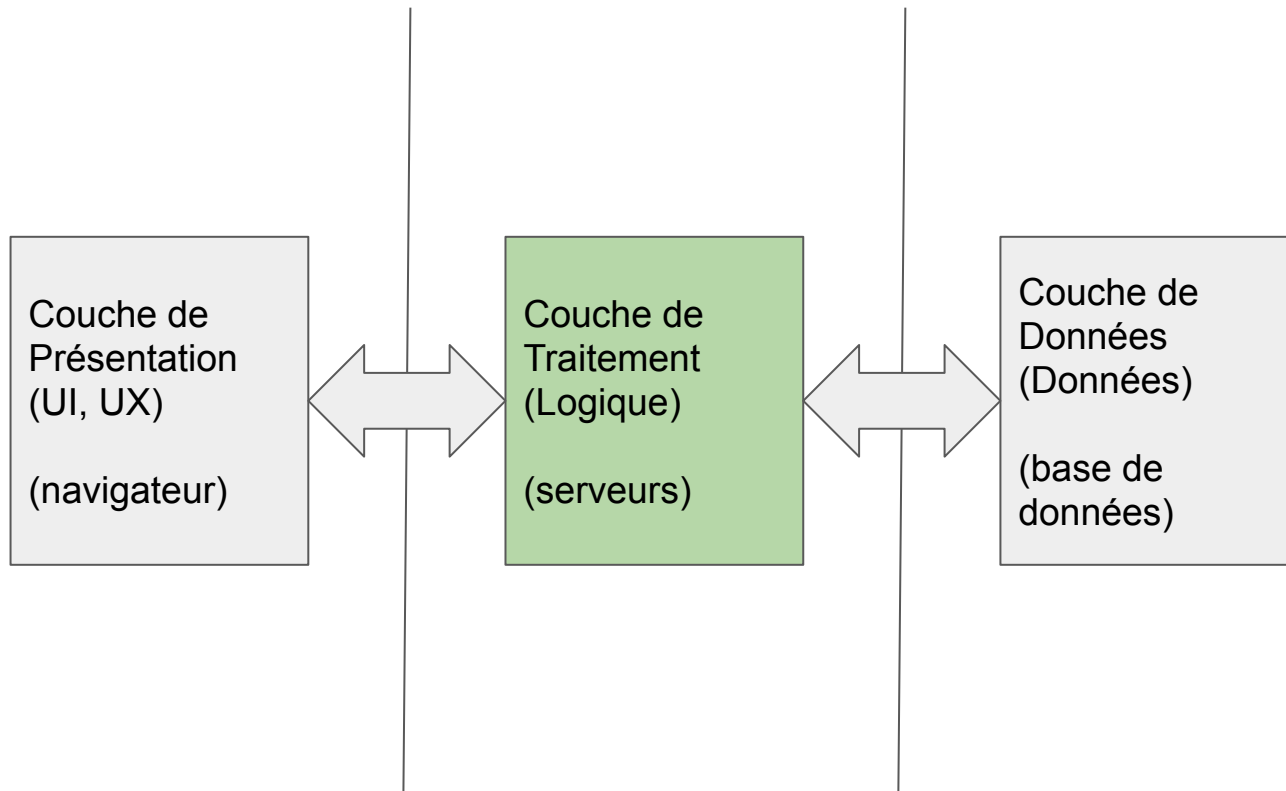
6

18

Badge
A small value indicator that can be overlaid on another object.

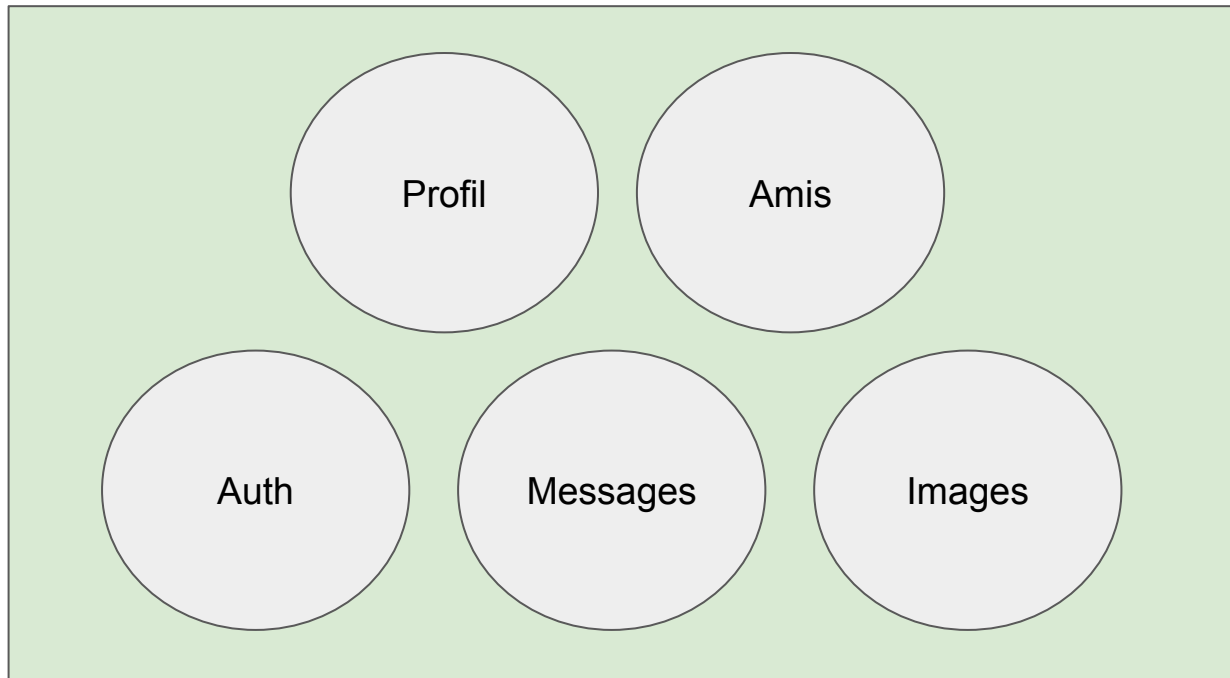
Exemple





Architecture monolithique

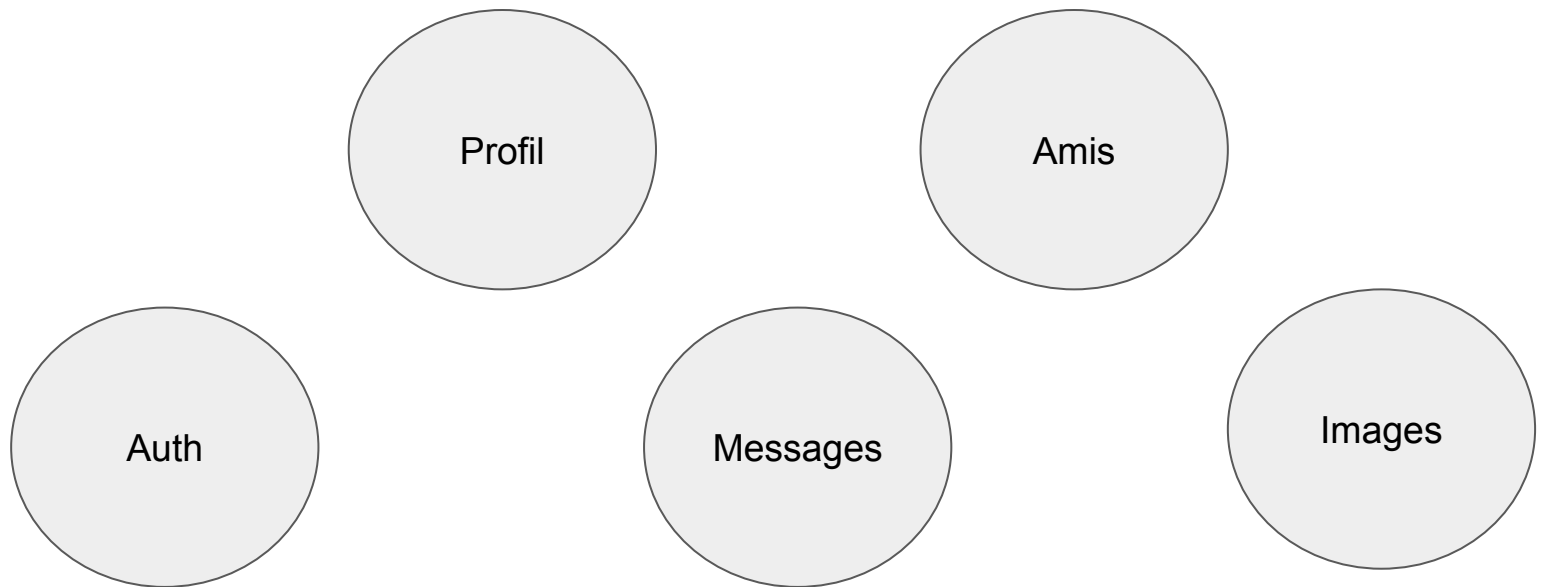
La couche de traitement consiste en un seul élément déployé. Cet élément contient l'ensemble des composants. Les composants (modules ou packages) communiquent par via appels de fonction.



Architecture par services

Un service est un composant déployable qui contient un ensemble de fonctionnalités qui sont exposé aux autres services via un API.

Dans une architecture par services, la couche de traitement consiste en plusieurs services interconnectés. L'exécution d'un scénario d'utilisation (use case) requiert la coordination d'un ou plusieurs services.



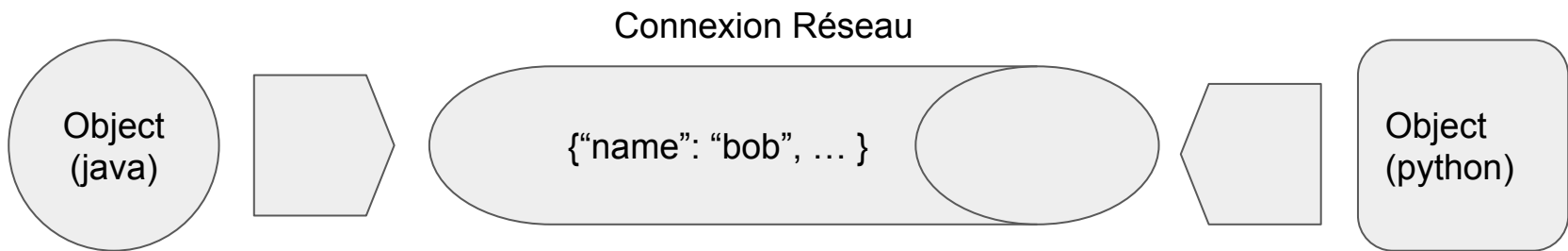
Communication inter-services

Il existe différents patrons de communication entre les services:

- requête-réponse (synchrone, e.g. HTTP)
- par messages (asynchrone, e.g. RabbitMQ)
- pair-à-pair (asynchrone, e.g. websocket, grpc*)

Sérialisation des données

Lorsque deux composants (possiblement écrits dans des langages différents) échangent des données, ils doivent s'entendre sur la représentation intermédiaire que ces données auront sur le réseau. Ainsi, lorsqu'un message est envoyé, le contenu (*payload*) doit être sérialisé par l'émetteur, puis dé-sérialisé par le receveur. Ceci assure l'interopérabilité entre les différents composants.



Qu'est-ce qu'une API?

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

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

En web, une API comporte

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

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

API RESTful en pratique

Typiquement, un API RESTful:

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

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

Opérations typiques - CRUD

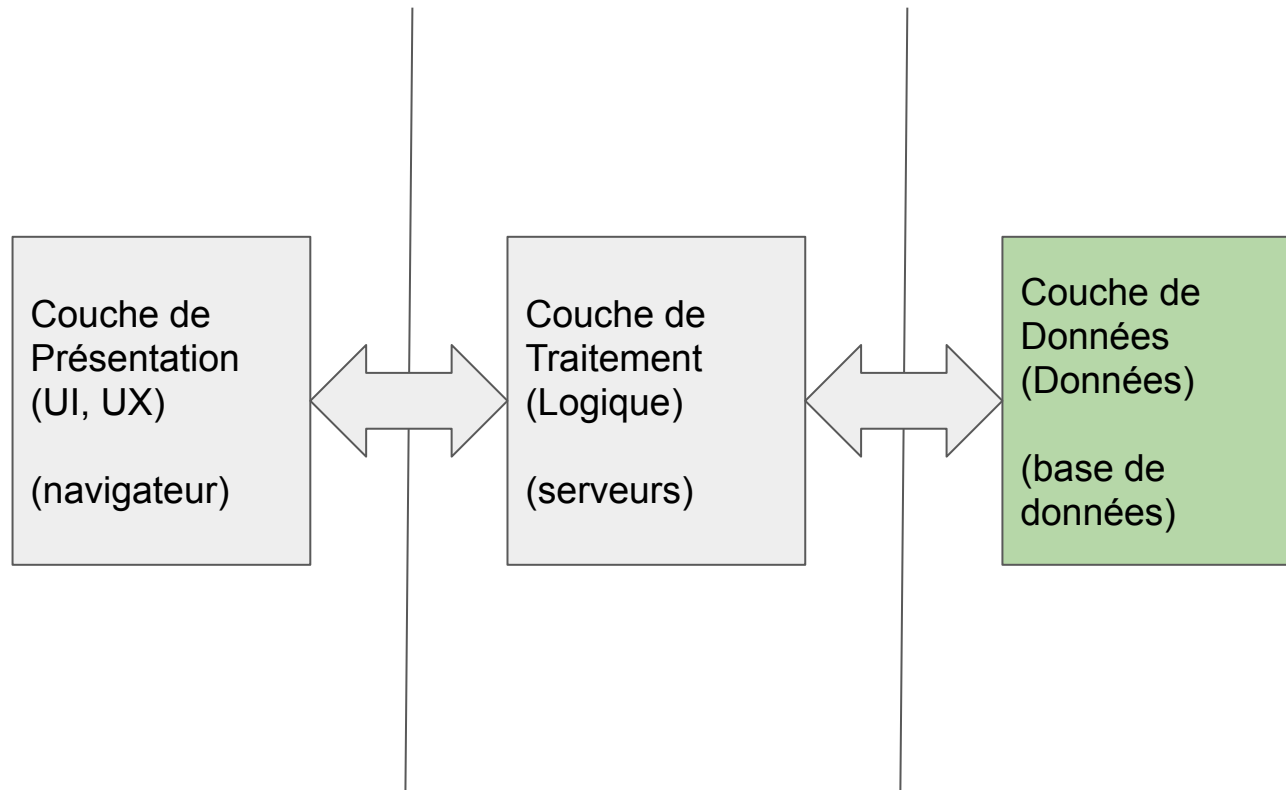
De manière générale, il est simple de modéliser une API qui offre les opérations de création, lecture, modification et suppression d'une ressource.

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

Ressources imbriquées

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

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



Base de données relationnelle (SGBDR)

Basé sur un **modèle de tables** possédant un **schéma** qui décrit les champs, les types et leurs contraintes.

Typiquement, chaque entité possède un **identificateur unique** (clé) qui est utilisé pour référencer un objet particulier dans la table.

Les relations sont définies en utilisant ces clés uniques.

Selon la cardinalité de la relation, elle peut être exprimée soit par une référence directe (un attribut qui réfère à une entité spécifique d'une autre table), soit par une table qui lie les clés de plusieurs enregistrements entre eux.

Bases de données de type clé-valeur

Base de données simple pour stocker des valeurs adressées par une clé (i.e. un dictionnaire).

Les données peuvent être en mémoire (type cache) ou persistées selon des règles (persistance périodique, après chaque opération, etc).

Le format de la clé est libre.

Le format des valeurs dépend de la technologie, mais la base de données peut accepter différents types et offrir différentes fonctionnalités selon le type (e.g. ajout d'un item à une liste de façon atomique).

Bases de données orientés documents

Les données sont stockées sous forme de **documents structurés** (style JSON) qui sont **groupés par collections** au lieu de tables.

Les documents d'une même collection n'ont pas nécessairement exactement les mêmes champs (hétérogènes).

Les documents sont accédés par le nom de la collection, puis la clé du document ou par une recherche (e.g. timestamp > 1664299885).

Les relations entre les entités sont définies par une référence vers un id d'un autre document comme dans un SGBDR, mais l'existence de la clé n'est pas nécessairement validée par la base de données.

Quelques bonnes vidéos sur les patrons de conception dans un monde non-relationnel: [NoSQL Data Modelling with Redis](#).

Modélisation - Insertion partielle

Pour optimiser les performances d'une application, il est aussi possible d'insérer (dupliquer) une partie des informations d'une autre entité dans l'entité principale.

En général, on choisit les informations nécessaires à l'affichage de l'entité principale et on va chercher les données complètes de l'entité secondaire seulement au besoin.

Avantages:

- Limite le nombre d'opérations lectures
- Plus rapide pour l'affichage

Désavantages:

- Duplication de donnée qu'il faut synchroniser
- Si la vue change, il est possible que le modèle ne soit plus aussi optimal

Authentication

Hachage et mots de passe

Comme l'application de la fonction de hachage est rapide et unidirectionnel, on applique la fonction de hachage au mot de passe que l'on peut alors stocker.

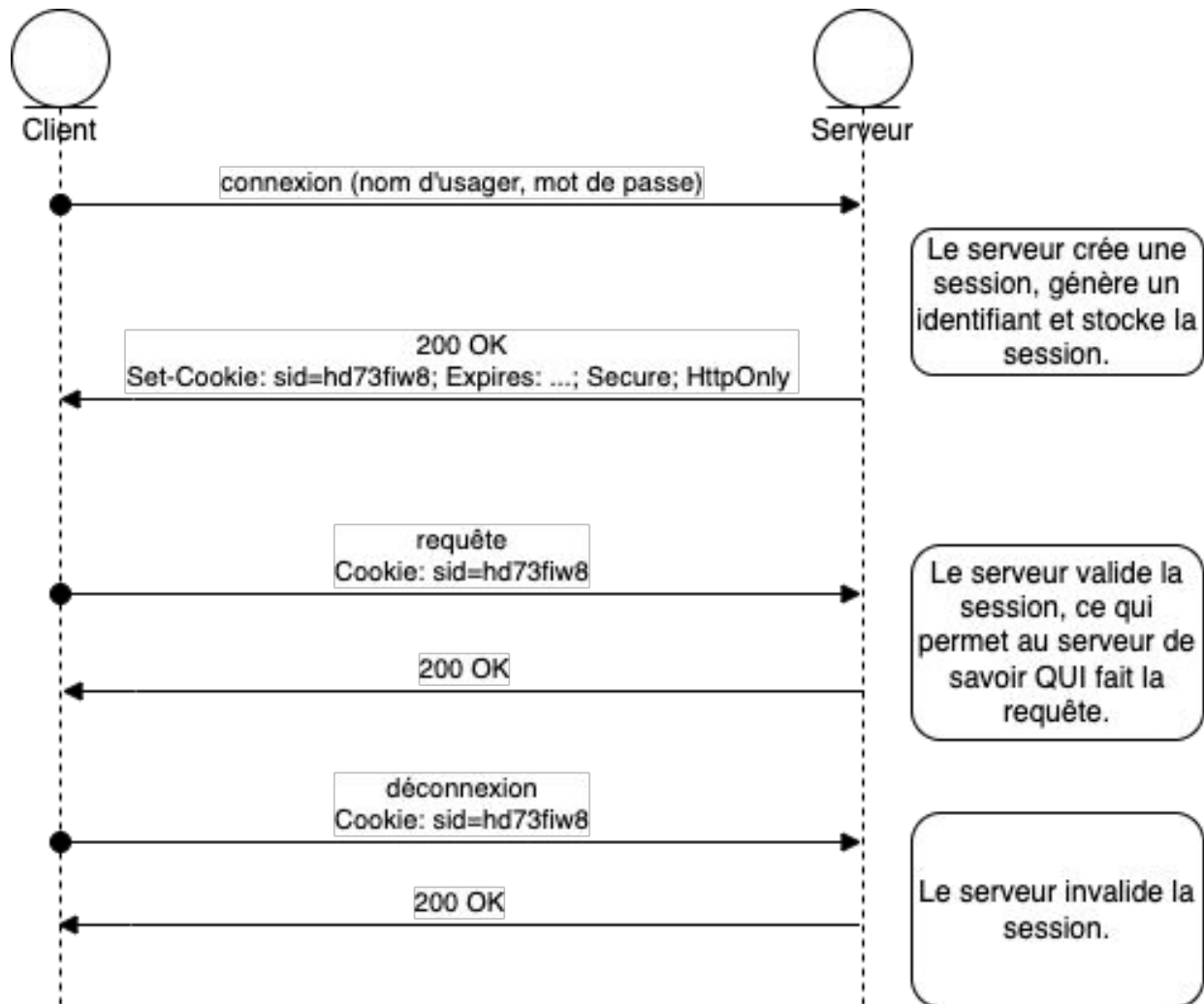
À partir de ce moment, il suffit d'appliquer à nouveau la fonction de hachage sur le mot de passe reçu lors de la connexion et de comparer avec la valeur stockée dans la base de données pour le valider.

Si jamais la base de données est compromise, seuls les *hash* seront visibles et comme il est *impossible* de retrouver les mots de passe à partir du *hash*, l'attaquant n'a pas accès aux mots de passe de vos utilisateurs (mais il a quand même eu accès à votre DB 😬).

Mécanisme d'authentification

Pour transmettre les informations d'authentification au serveur, il existe différentes options:

- l'en-tête HTTP Authorization
 - Basic: le nom d'utilisateur et le mot de pass est retransmis à chaque requête
 - Bearer: le jeton (token) est retransmis à chaque requête
- les cookies HTTP
 - les cookies sont initialement créés par le serveur
 - les cookies sont retransmis à chaque requête pour la même origine



JSON Web Token - JWT (jot?!)

Un autre mécanisme populaire pour gérer l'authentification est le JSON Web Token (JWT). Contrairement au mécanisme précédent utilisant les cookies de session, le serveur n'a pas à créer ni conserver de session.

Le JWT contient l'ensemble des informations nécessaires, incluant l'identifiant de l'utilisateur.

Le jeton est signé par le serveur.

Le client renvoie le jeton dans ses requêtes (en-tête `Authorization`, cookie).

Le serveur peut alors valider la signature pour s'assurer que le jeton n'a pas été modifié.

Mécanisme de signature

Pour permettre au serveur de vérifier que le jeton reçu du client n'a pas été altéré, la troisième partie du jeton est une signature cryptographique.

`Algo (base64UrlEncode (header) + "." + base64UrlEncode (payload) , secret)`

Différents algorithmes peuvent être utilisés. Soit des fonctions de hachage pour générer un HMAC, soit un algorithme de cryptage asymétrique.

<https://jwt.io/>

JWT bonnes pratiques

Attention! Le contenu du JWT n'est qu'encodé donc n'importe qui peut lire l'information qu'il contient, sauf s'il est crypté avant d'être encodé en Base64Url.

Il est donc important de ne pas mettre des informations sensibles dans le contenu du jeton.

Pour transmettre un JWT on peut utiliser soit l'en-tête `Authorization` ou les cookies.

Si on utilise les cookies, on veut utiliser les attributs `Secure` et `HttpOnly` pour éviter qu'un scripte malhonnête n'accède au jeton et l'utilise pour effectuer des requêtes.

Si on utilise l'en-tête `Authorization`, il est préférable de garder le JWT en mémoire.

Il est possible de le garder dans le `localStorage` mais cela le rend vulnérable à certains types d'attaques.

Gestion d'erreurs

Validation de formulaire

Lorsqu'on demande à l'utilisateur de remplir un formulaire web, il faut faire attention aux erreurs de saisie et aux données invalides.

On peut **limiter les erreurs en ajoutant des règles de validation** aux formulaires web. Si les valeurs entrées par l'utilisateur ne sont pas conformes aux règles de validation, les données sont refusées jusqu'à ce que l'utilisateur les corrige.

Ceci évitera de traiter ou stocker des données incomplètes ou invalides.

Validation de requêtes

Côté serveur, il est préférable de **valider les données reçues (corps HTTP, paramètres de requête, etc) aussitôt possible**. Ceci permet de détecter les erreurs rapidement et d'assurer aux composants suivants que les données qu'ils recevront seront valides.

Si les données sont invalides, le **serveur doit répondre avec un code d'erreur** approprié (typiquement dans la famille des 4xx).

Il est aussi possible d'ajouter un code et un message d'erreur spécifique à l'application dans le corps de la réponse.

Contrairement aux erreurs de la couche de présentation, ces erreurs seront interprétées par une autre service web ou par l'application frontend et **ne devraient pas être visibles des utilisateurs**.

Journaux - Niveaux

La sélection du bon niveau pour une entrée dans les journaux est très importante. En mode production, pour des raisons de performance, seul les niveaux supérieurs seront activés.

Donc si une erreur importante est inscrite avec un niveau trop bas, elle ne sera pas visible. À l'inverse, trop d'entrées sans importance dans le journal d'un serveur ne font qu'ajouter du bruit (et déclencher la rotation plus rapidement).

Niveau	Description
Trace / Debug	Information pour déboguer l'application en mode développement.
Info	Événements significatifs, mais normaux.
Warn	Information décrivant un état qui est potentiellement problématique.
Error	Erreur importante, mais qui n'empêche pas l'application de fonctionner.
Fatal	Erreur critique. L'application ne fonctionne plus ou en mode dégradé.

Journaux - bonnes pratiques

- format identique de log dans tous vos services
- ajouter les traces d'appel (*stacktraces*)
- *grepable* - information essentielle sur une seule ligne ou *parsable* - json
- ajouter l'id du thread (dans un serveur multi-threaded)
- ajouter l'id de corrélation de la requête (système multi-services)
- choisir le bon niveau de log (voir page suivante)
- rotation des journaux (pour éviter de remplir le disque)
- centraliser les journaux (pour faciliter le débogage)

Traces distribuées

Dans une architecture par services, il est souvent utile de pouvoir suivre une requête du client au travers des différents services.

De la même façon qu'il est possible d'insérer l'id d'un thread, il est aussi possible d'insérer un id de corrélation unique (GUID) qui sera généré à l'entrée du système et ajouté à chaque appels d'API subséquents (via les en-têtes HTTP). Si l'id de corrélation est ajouté à toutes les entrées de journaux, il sera possible de suivre l'évolution de la requête dans le temps à travers les journaux des différents serveurs.

Ce sera encore plus utile si combiné avec un système d'agrégation de journaux qui centralise l'ensemble des journaux dans un engin de recherche pour faciliter le suivi.

Résilience

Délai d'inactivité (Timeout)

Une bonne pratique à utiliser lorsqu'on fait un appel à un API, est de toujours spécifier un délai maximal au-delà duquel, le client agira comme si la requête avait échoué.

Ceci permet de libérer des ressources rapidement (important en backend) et de ne pas laisser l'utilisateur devant une page qui semble se charger indéfiniment.

Reprise (Retry)

Pour certains types d'erreurs qui peuvent être temporaires, la stratégie de reprise est efficace.

Par exemple, un client HTTP pourrait réessayer si

- le délai d'inactivité (timeout) est expiré (serveur inactif)
- sur un code 408 (request timeout), 429 (too many requests), 500 (internal server error), 502 (bad gateway), 503 (service not available), 504 (gateway timeout).

Dans les autres cas, il est souvent inutile de réessayer. Par exemple, sur un code 400 (bad request), il est peu probable que le serveur accepte la même requête la 2^e fois.

Redondance

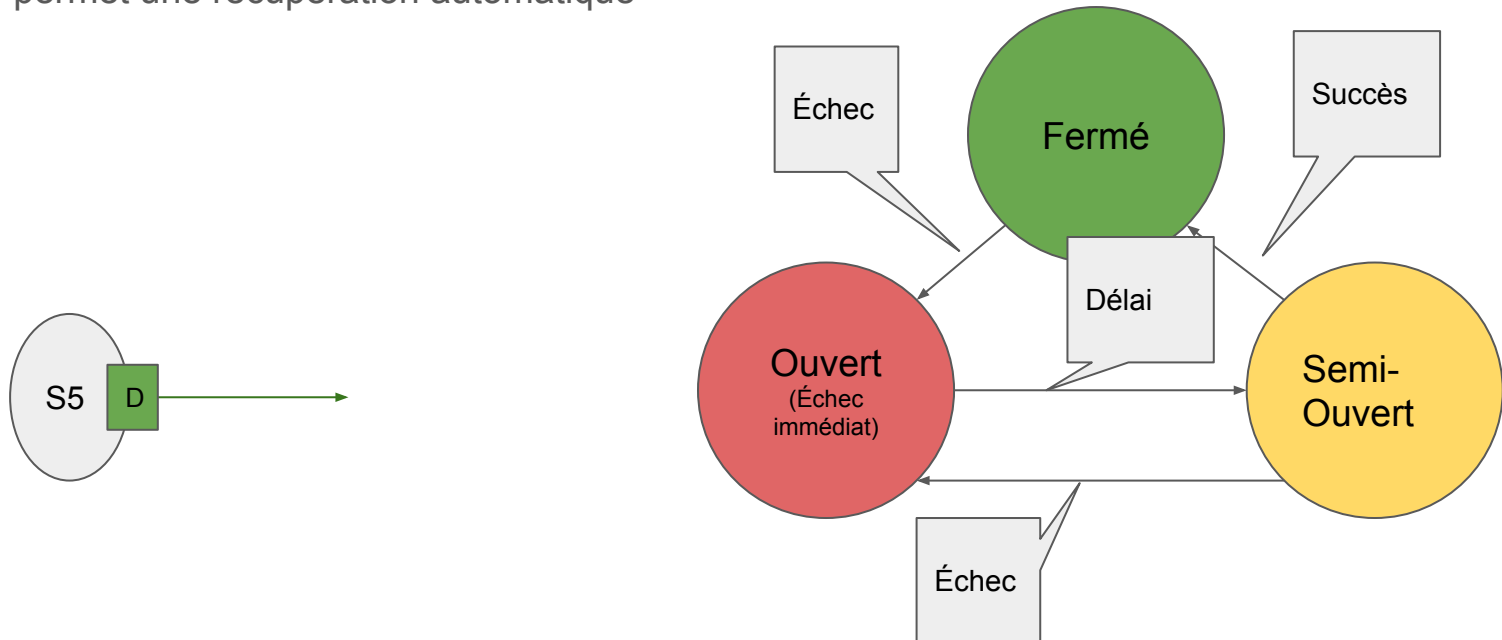
Une stratégie simple consiste à intentionnellement créer de la redondance (plusieurs instances du même service) dans le système. Cette stratégie requiert l'utilisation d'un balancer de charge qui distribue les requêtes entre les multiples instances d'un même service.

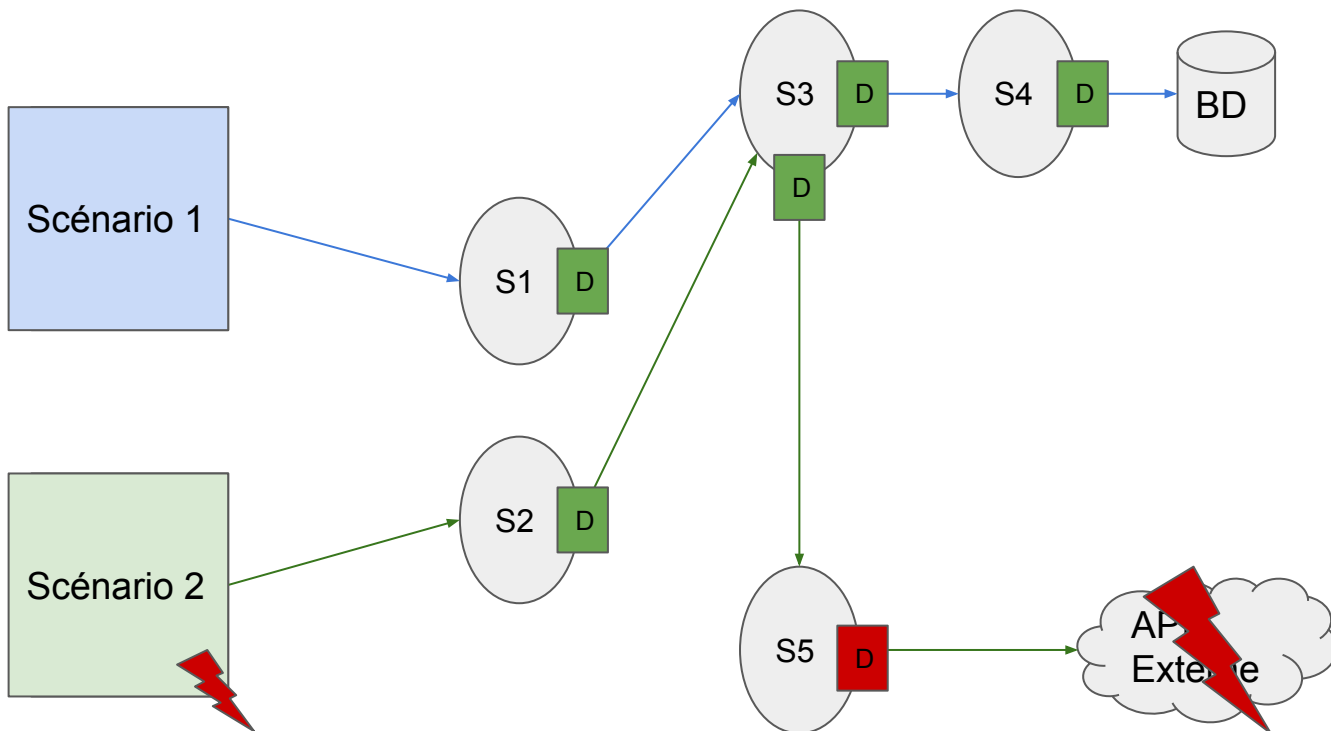
Par exemple, si, pour la charge sur notre système, nous avons besoin de 2 instances d'un service spécifique, nous allons en déployer 3.

De cette façon, si un serveur tombe en panne, l'application pourra continuer de fonctionner le temps que la panne soit réglée.

Solution: Disjoncteur (Circuit Breaker)

- Mécanisme utilisé pour limiter les impacts d'un problème et éviter une défaillance en cascade.
- Un outil simple qui protège un composant des échecs de ses dépendances (filtre).
 - limite les impacts de la défaillance
 - évite de surcharger un composant instable
 - permet une récupération automatique





Post-mortem

Lorsqu'un problème survient dans un environnement de production et que ce problème a de graves conséquences (temps de panne, perte de données, etc), il est important de bien en analyser les causes pour éviter que le même problème survienne à nouveau.

Souvent, dans le feu de l'action, la première étape consiste à trouver une solution à très court terme (workaround) qui ne demande pas ou très peu de changements.

Une fois le problème sous contrôle, la deuxième étape consiste à trouver une solution à long terme (un vrai correctif) pour le problème en question.

Enfin, il convient de comprendre pourquoi ce problème est survenu, identifier la cause et de déterminer les actions à prendre pour la suite. Par exemple, si c'est une erreur humaine qui a déclenché la panne, est-il possible d'automatiser cette partie du processus?

Sécurité

XSS - cross-site scripting - injection de script

L'attaque XSS consiste à **injecter un code malicieux dans la page web**, ce qui permet à l'attaquant d'utiliser le contexte de la page pour effectuer des actions.

L'attaquant n'a pas besoin de modifier le code de l'application web. Il profite des champs d'entrées (inputs) et y insère des valeurs spécifiques qui injectent le script lorsqu'une page particulière est affichée. L'attaquant profite alors du fait que le script s'exécute dans le navigateur de l'utilisateur en utilisant le contexte de celui-ci (cookies, permissions, etc).

Une application est vulnérable aux attaques de type XSS lorsqu'elle permet aux utilisateurs d'entrer de l'information qui sera ensuite ajoutée à la page sans validation. Par exemple, si une application permet d'entrer des balises HTML et qu'elle honore ensuite ses balises dans l'affichage, elle est probablement vulnérable aux attaques XSS.

CSRF - cross-site request forgery

Ce type d'attaque concerne un utilisateur qui active un **lien malicieux** qui exécute une opération avec ses privilèges.

L'attaquant, qui connaît ou a déduit l'API de l'application web, fournit un lien à la victime.

Lorsque la victime clique sur le lien, elle déclenche une requête (`GET`) du navigateur web qui transmet les informations de session de l'utilisateur au serveur. **L'opération s'exécute alors sans son consentement, mais avec ses privilèges, car le navigateur va faire suivre les cookies.**

L'attaquant peut aussi créer un faux formulaire web sur un site frauduleux qu'il contrôle et qui déclenche un `POST` avec les informations entrées par l'utilisateur.

Protection contre les CSRFs - Jetons

Pour protéger contre les CSRFs, il est possible d'utiliser un jeton de synchronisation. Le serveur génère un jeton (aléatoire, par session ou par requête) qui est retourné au client dans la page HTML ou dans la réponse JSON.

Ce jeton doit être envoyé au serveur lors de l'opération `POST` (formulaire) soit comme paramètre, soit dans une en-tête HTTP.

Par exemple pour un site dynamique: le HTML généré contient le jeton:

```
<input type="hidden" name="csrfmiddlewaretoken"
value="KbyUmhTLMpYj7CD2di7JKP1P3qmLlkPt" />
```

L'attaquant ne pourra pas avoir le bon jeton dans son faux formulaire et son `POST` sera rejeté par le serveur.

Protection contre les CSRFs - Jetons

Pour une application monopage, le jeton peut être transmis dans une réponse d'API du serveur et stocké en mémoire.

Lors de l'appel à l'API, le code javascript de l'application ajoute le jeton dans une en-tête HTTP personnalisée.

Le serveur valide ensuite l'en-tête avant d'effectuer l'opération pour s'assurer que la requête est légitime.

https://cheatsheetseries.owasp.org/cheatsheets/Cross-Site_Request_Forgery_Prevention_Cheat_Sheet.html#synchronizer-token-pattern

Protection contre les CSRFs - Option sans état

Pour éviter de maintenir un état dans l'application pour valider les jetons de synchronisation, il est possible d'utiliser la stratégie du double submit cookie.

Dans ce cas, le serveur crée un jeton aléatoire et l'ajoute comme cookie (e.g. XSRF-TOKEN).

Lorsqu'une requête légitime est envoyée, le code de l'application ajoute la valeur du cookie dans un en-tête particulier (e.g. X-XSRF-TOKEN).

Le serveur valide donc que la valeur du cookie et de l'en-tête son identique.

À noter qu'il n'est pas possible pour un script de lire les valeurs des cookies provenant d'une autre origine.

Injection SQL

L'injection SQL consiste à construire une valeur particulière pour un entrée (input) de formulaire, sachant que cette valeur sera utilisée pour bâtir une requête SQL dans le serveur.

Exemple: Si le code ressemble à

```
sqlQuery = 'SELECT * FROM users WHERE USER = ' + user_id
```

et qu'il est possible pour l'attaquant de construire `user_id`.

Que se passe-t-il si `user_id` correspond à

- `'1=1' -> SELECT * FROM users where USER = true`
- `'123abc; DROP TABLE users;' -> SELECT * FROM users WHERE USER = 123abd; DROP TABLE users;`
- `'123abc; UPDATE users SET credits = 10000 WHERE user = 123abd;'`
`->$$$`

Partage des ressources entre origines multiples - CORS

La politique de même origine (Same Origin Policy) empêche les scripts de charger du contenu à partir d'un serveur d'une origine différente via `XMLHttpRequest` ou `fetch`. L'origine étant définie par le protocole, le domaine et le port.

Ceci permet d'éviter qu'un site malveillant accède aux ressources d'une autre origine (e.g. `api.bank.com`) en envoyant les cookies d'une victime (pour `api.bank.com`).

Le navigateur effectue la requête, mais ne considère pas la réponse sauf si le serveur permet les origines multiples (`Access-Control-Allow-Origin`) et que l'origine de la requête est acceptée par le serveur. Dans le cas mentionné plus haut, le navigateur va simplement refuser la réponse à la requête.

Par contre, il est possible que le site `bank.com` (propriétaire de `api.bank.com`) ait besoin de faire des requêtes à `api.bank.com`. Donc les serveurs d'`api.bank.com` accepteront les requêtes ayant comme origine `bank.com`.

Tests

Types de tests

Par ordre de taille des composants testés:

- tests unitaires
- tests d'intégration
- tests de bout en bout
- tests d'acceptation
- tests de fumée
- tests de performance
- tests système

Infrastructure et déploiement

Environnements

Le processus de développement d'une application web requiert différents environnements qui ont chacun des caractéristiques et des objectifs particuliers.

Les environnements typiques sont:

- développement
- test
- mise en scène
- production

Machine virtuelle

Pour pouvoir créer plusieurs machines virtuelles distinctes et isolées les unes des autres, il est possible d'introduire un logiciel appelé **hyperviseur** qui offre une abstraction du matériel.

Il existe 2 type d'hyperviseurs:

- Type 1 - *Bare Metal* - S'exécute directement sur le matériel (e.g. Oracle VM Server, Xen)
- Type 2 - *Hosted* - S'exécute dans un OS (e.g Oracle VirtualBox, VMware Workstation)

Chaque machine virtuelle s'exécute *sur* l'hyperviseur et possède ses propres ressources virtuelles (CPU, RAM, etc). Elle ne peut pas accéder aux autres machines virtuelles présentes sur la même machine physique.

Chaque machine virtuelle peut exécuter un système d'exploitation différent.

Les machines virtuelles peuvent aussi être copiées ou transférées d'un hôte à l'autre.

Conteneur

Un système de conteneurs utilise une abstraction au niveau du système d'exploitation (qui est partagé par tous les conteneurs).

Le conteneur contient alors seulement l'application à exécuter et l'ensemble de ses dépendances.

L'image d'un conteneur est beaucoup plus petite comparé à une machine virtuelle qui contient l'OS.

Pour une architecture par services, l'utilisation de conteneur permet de réduire de beaucoup les ressources nécessaires.

Si le conteneur est utilisé comme unité de déploiement, un développeur peut exécuter plusieurs conteneurs en parallèle pour avoir une version locale du système complet.

Image vs Conteneur

Une **image** est une définition immuable qui sert de gabarit pour créer les conteneurs.

Un **conteneur**

- est une instance exécutable d'une image (avec un nom)
- il est possible de créer, démarrer, arrêter, déplacer et effacer un conteneur
- un conteneur peut s'exécuter localement, sur une machine virtuelle ou déployé dans le cloud
- un conteneur est isolé des autres conteneurs (namespace, cgroup)

Infrastructure en tant que service (IaaS)

Dans un modèle d'infrastructure en tant que service (Infrastructure as a Service), le fournisseur gère le réseau et la partie matériel (machine et disk, incluant la virtualisation) et, parfois, le système d'exploitation des serveurs.

Vous n'avez donc qu'à gérer la couche de logiciel spécifique à votre application web: serveur web, environnement d'exécution, etc.

Exemple: AWS, Azure, Google Compute Engine

Plateforme en tant que service (PaaS)

Dans une solution de plateforme en tant que service (Platform as a Service), le fournisseur gère tout sauf votre application.

Une solution PaaS peut même effectuer la mise à l'échelle automatiquement.

Exemple: AWS Elastic Beanstalk, Google App Engine

Fonction en tant que service (FaaS)

Dans une solution de fonction en tant que service (Function as a Service), le fournisseur gère tout et vous n'avez qu'à déployer des morceaux de code qui seront exécutés selon certains déclencheurs (appel HTTP, cron, modification dans firestore, etc).

La solution FaaS effectue la mise à l'échelle automatiquement.

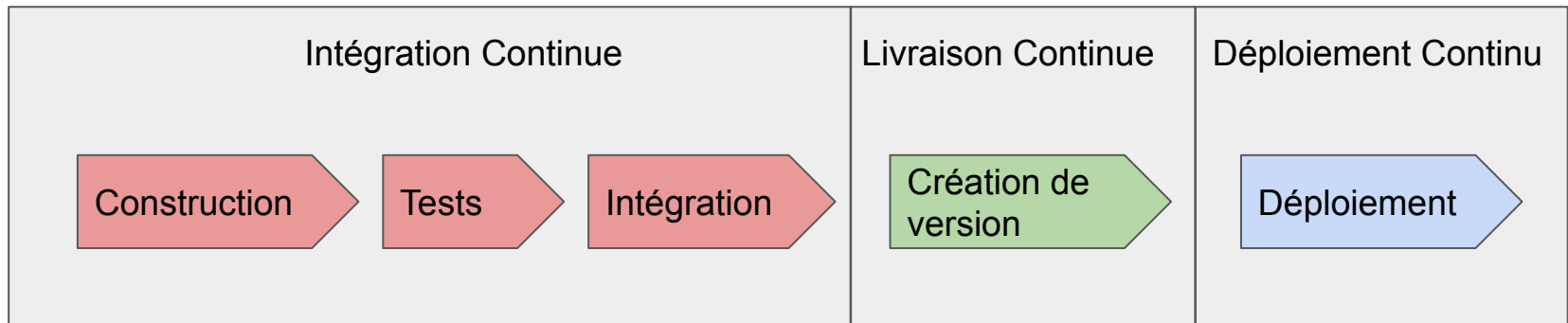
Exemple: AWS Lambda, Google Cloud Functions

Logiciel en tant que service (SaaS)

Pour une solution de logiciel en tant que service (Software as a Service), le fournisseur gère la gestion de l'ensemble du matériel et du logiciel.

Il suffit ensuite de s'y connecter (e.g. API, base de données) ou simplement d'utiliser le logiciel (e.g. une application de gestion de site web).

Pipeline CI/CD



Maintenance et évolution

Maintenance

La maintenance est l'étape du processus de développement logiciel où on modifie une application après qu'elle ait été mise en service.

Habituellement, la maintenance n'implique pas de changement majeur à l'architecture du système. Elle consiste plutôt en une évolution graduelle réalisée en modifiant ou en ajoutant des composants.

Les activités de maintenance sont:

- Correction de défauts
- Adaptation de l'application à un environnement d'opération en évolution
- Ajout ou ajustement de fonctionnalités

Dette technique

*La dette technique peut être accrue lors d'un codage non optimal. **Une conception logicielle négligée induit des coûts futurs**, les intérêts, à rembourser sous forme de temps de développement supplémentaire, et des bugs de plus en plus fréquents. La dette technique doit être remboursée rapidement pour éviter l'accumulation de ces intérêts, d'où l'analogie avec le concept de dette financière.*

- https://fr.wikipedia.org/wiki/Dette_technique

Il arrive souvent que les gens moins techniques ne comprennent pas les implications d'une solution rapide sous-optimale pour ajouter une fonctionnalité ou corriger un défaut dans une application.

Bien qu'elle ne soit pas chiffrée, le concept de dette technique est une métaphore qui exprime bien l'impact de différentes décisions techniques.

Métriques

Savoir qu'un service est fonctionnel ou non est une bonne première étape pour surveiller un système.

Il est encore plus intéressant d'en connaître davantage sur l'état du service.

Pour se faire, il est possible d'exposer des métriques qui seront publiées pour être stockées et consultées.

Il existe différentes approches:

- le service publie ses métriques à interval régulier
- le service expose ses métriques et un système externe vient le lire à intervalle régulier

Les types de métriques

Compteur: Le nombre d'événements (ex. nombre d'appels). La valeur du compteur augmente toujours (sauf au démarrage).

Jauge: Une mesure instantanée (ex. taille de la heap). La valeur de la jauge peut augmenter ou diminuer.

Histogramme: Une répartition des différentes valeurs enregistrée avec calcul de percentiles (ex. taille des messages).

Chronomètre: Mesure un temps d'exécution et le nombre d'appels (ex. temps de réponse).

Le **taux** est souvent calculé par l'agrégateur de métriques en fonction d'un intervalle de temps spécifié.

Par exemple, le taux de requêtes par seconde est calculé à partir d'un **compteur** de requêtes en divisant la variation du compteur pour un intervalle de temps donné (ex. 10s). Donc si le compteur a augmenté de 100 en 10s, le taux est de 10 requêtes par seconde.

Les signaux d'or

Selon le guide de l'ingénierie de la fiabilité des sites de Google (<https://sre.google/>), il faut observer au minimum les 4 signaux suivant:

- le temps de réponse (séparer succès et échecs)
- la charge (requêtes/seconde)
- le taux d'erreur
- la saturation - % de l'utilisation maximale du système

Alertes

Pour s'assurer de détecter et de réagir rapidement lorsqu'une défaillance survient, il est souhaitable de définir une règle automatique qui déclenche une alerte lorsqu'une condition particulière survient (habituellement une métrique qui dépasse un seuil).

L'objectif de l'alerte est d'avertir l'équipe responsable que le système est dans un état qui demande une intervention.

Pour éviter la fatigue (qui conduit à ignorer les alertes), il est important que:

- la ou les actions à prendre pour chaque alerte soient définies
- une alerte doit requérir une action *intelligente* (sinon automatiser la réponse)
- une alerte doit signaler un nouveau problème (pas d'alerte redondante)

Observabilité

L'observabilité est une caractéristique d'un système qui permet d'inférer son état interne à partir des données qu'elle expose.

Les 3 piliers de l'observabilité:

- journaux
- métriques
- traces

La combinaison des trois éléments permet souvent de diagnostiquer les défaillances.

Il est important, lorsque l'on bâtit un système, de penser à l'**observabilité** et de ne pas hésiter à ajouter des métriques et des événements dans les journaux.

Rétrocompatibilité

Comme il est impossible de tout mettre à jour instantanément (application monopage, services, etc), la séquence de déploiement doit être une suite de changements rétrocompatibles.

Comment faire pour effectuer une modification a priori non rétrocompatible sans attendre une nouvelle version majeure?

Exemple:

Comment ajouter un champ obligatoire dans le modèle de données partagé entre plusieurs composants (ex. application monopage, services, base de données)?

À la fin du processus de mise à jour, tous les enregistrements (passés et futurs) devront contenir une valeur valide.

Pensez en termes de *lecture* et d'*écriture*.

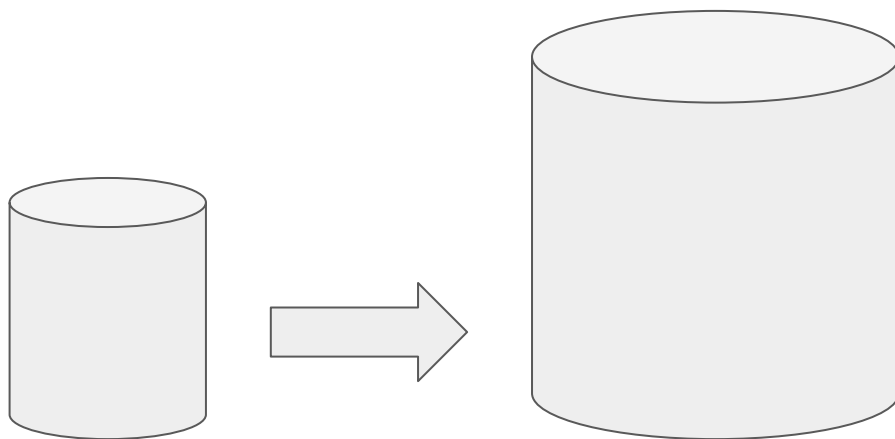
Mise à l'échelle

Mise à l'échelle verticale

La mise à l'échelle verticale consiste à augmenter la capacité de chacun des éléments existants du système.

Par exemple, on peut augmenter la puissance des CPUs, ajouter de la RAM, etc.

Il est aussi possible d'augmenter la capacité en optimisant l'utilisation des ressources (ex: optimisation de code, compression de données, etc).

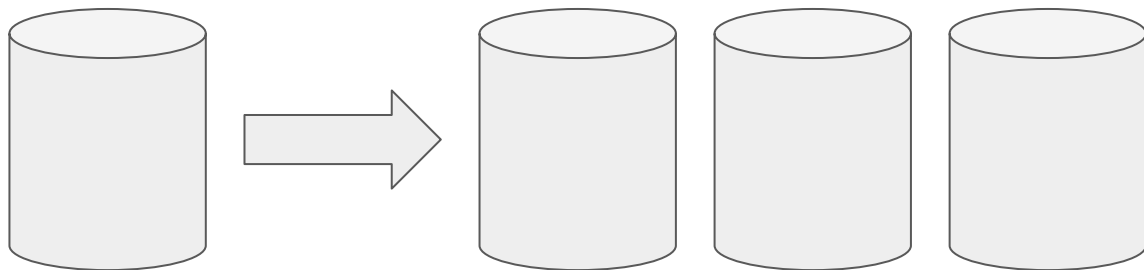


Mise à l'échelle horizontale

Une autre façon d'ajouter de la capacité est d'ajouter des nouvelles instances de composants (serveurs ou conteneurs) qui se partagent la charge.

Ce type de mise à l'échelle rend aussi le système plus résilient, mais plus difficile à opérer et surveiller.

Pour distribuer la charge équitablement, on ajoute un balancer de charge (load balancer).



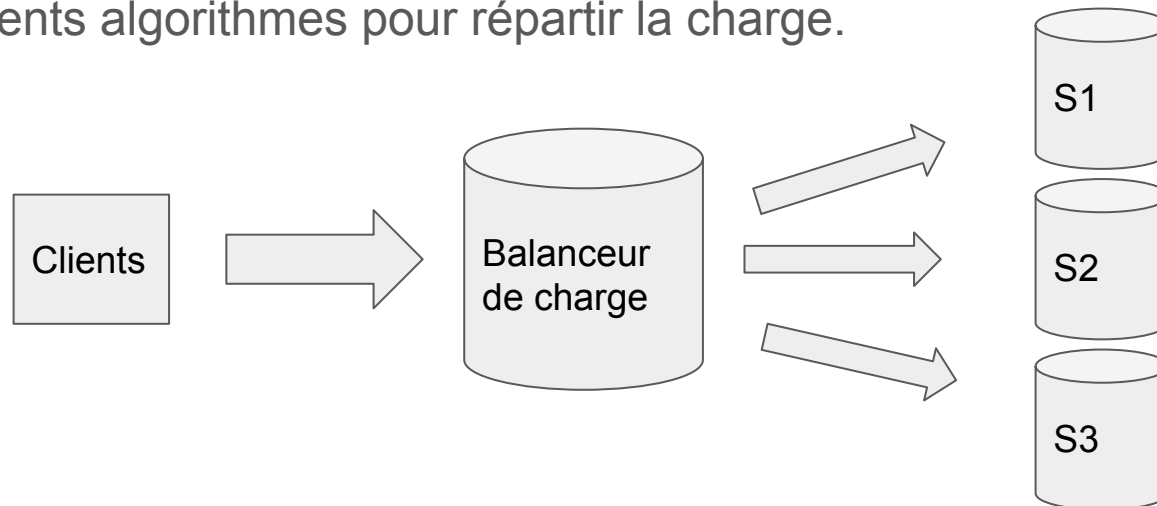
Balanceur de charge

Le rôle du balanceur de charge est de **répartir la charge entre différents composants (serveurs, conteneurs) qui ont un rôle identique** (ex: un service).

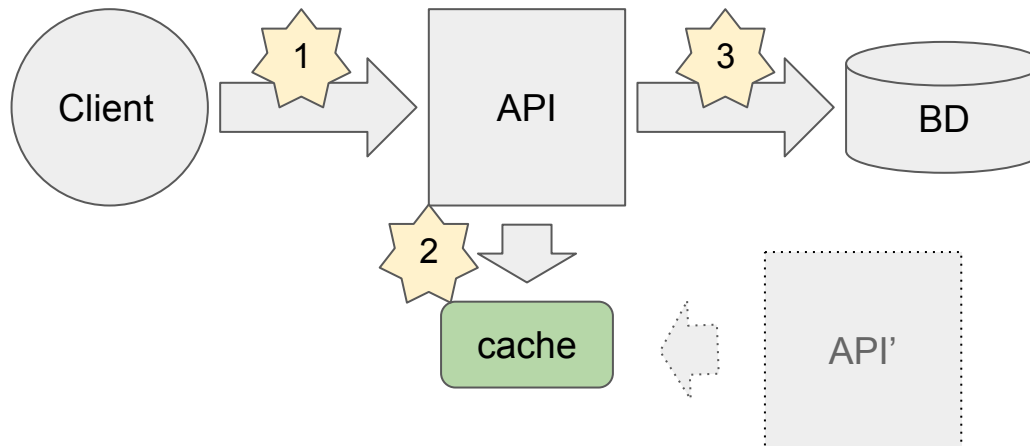
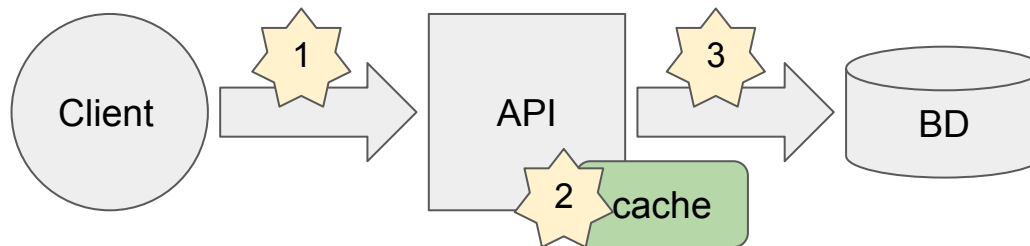
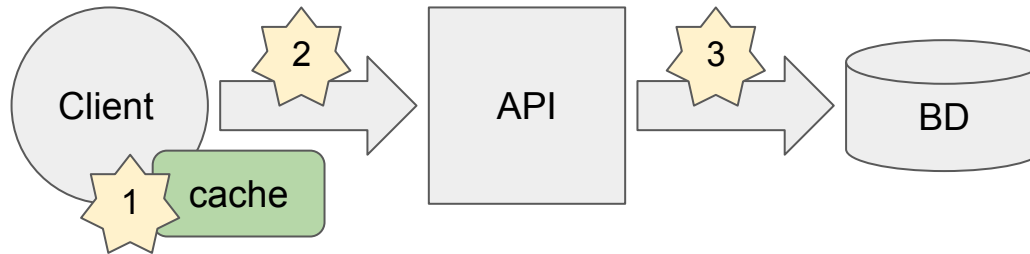
Les clients contactent le balanceur de charge et non les instances individuelles. Ceci permet de modifier dynamiquement le nombre d'instances sans avoir à modifier la configuration des clients.

Le balanceur de charge peut être matériel ou logiciel.

Il existe différents algorithmes pour répartir la charge.



Cache - Exemples



Base de données et mise à l'échelle

Jusqu'à maintenant nous avons surtout considéré la mise à l'échelle horizontale de composants dits *sans état*.

Pour ce type de composant, il suffit d'ajouter des instances pour augmenter la capacité car chaque instance est identique.

Pour une base de données, lorsqu'on atteint une limite (CPU, espace disque, taille de table) on ne peut pas seulement ajouter des nouveaux serveurs, il faut considérer comment les données existantes seront partagées.

Il existe plusieurs stratégies. Nous en examinerons deux:

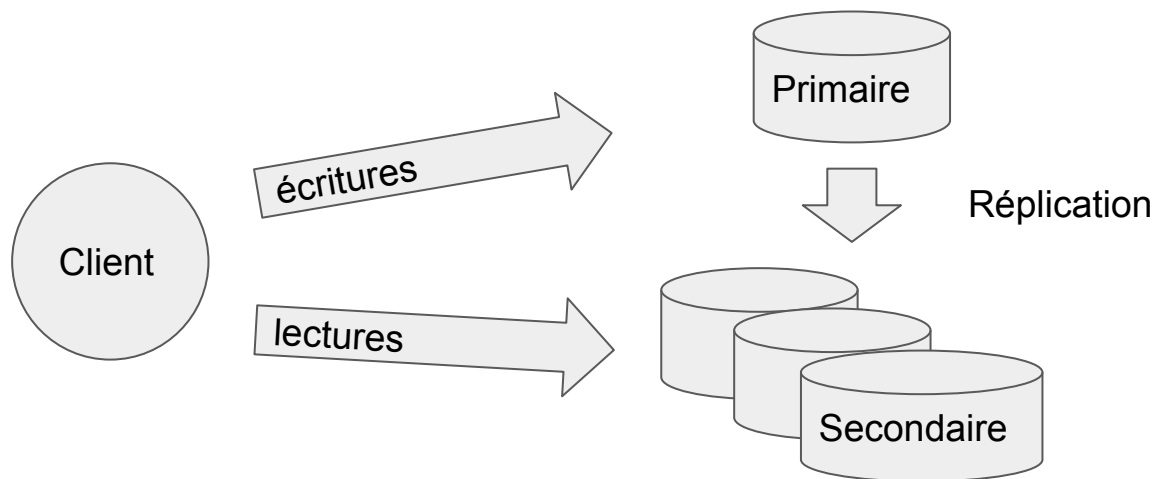
- Réplicats de lecture
- Partitionnement

Réplicats de lecture

Si la limitation de la base de données est liée aux nombres d'appels en lecture, il est possible d'ajouter des réplicats dédiés aux opérations de lecture.

Un serveur devient le serveur primaire et les autres, les serveurs secondaires.

Les opérations d'écriture sont seulement effectuées sur le serveur primaire.
Les données sont répliquées vers les serveurs secondaires qui se partagent la charge d'opérations de lecture.



Partitionnement

Si la limitation du serveur de base de données est lié à la quantité de données, il est possible de partitionner (diviser) les données pour ensuite les partager sur plusieurs serveurs.

L'algorithme de partitionnement doit être déterministe: à partir d'une information (clé, valeur d'un champ) on doit être en mesure de retrouver la partition.

Exemple MySQL, création d'une table avec 10 partitions:

```
CREATE TABLE userslogs (  
    username VARCHAR(20) NOT NULL,  
    logdata BLOB NOT NULL,  
    created DATETIME NOT NULL,  
    PRIMARY KEY(username, created)  
)  
PARTITION BY HASH( TO_DAYS(created) )  
PARTITIONS 10;
```

Types de partitionnement

Il existe différentes stratégies de partitionnement:

- **Intervalle:** On choisit un champ et on assigne les partitions en fonction d'intervalle de valeur.
 - Par exemple: P1: année < 2015, P2: année = 2016, ..., Pn: année > 2022
- **Liste:** On choisit un champ et on assigne les partitions selon la valeur du champ.
 - Par exemple: P1: catégorie = [A, B, C], P2: catégories = [D, F]
- **Clé:** On utilise la clé de l'enregistrement pour partitionner la table en divisant l'ensemble des clés possibles en partitions de tailles équivalentes.
 - Par exemple: clé % nb partitions
- **Hash:** On applique une fonction de hachage sur un ou plusieurs champ pour déterminer la partition (les champs doivent être connus du client!).
 - Par exemple: hash(nom) % nb partitions

Une bonne stratégie de partitionnement distribue les données de façon équitable entre les différentes partitions pour éviter un déséquilibre en termes de taille et d'utilisation.

Application web progressive

Application web progressive

*Une progressive web app (PWA) est une **application web qui consiste en des pages ou des sites web, et qui peuvent apparaître à l'utilisateur de la même manière que les applications natives ou les applications mobiles.** Ce type d'application tente de combiner les fonctionnalités offertes par la plupart des navigateurs modernes avec les avantages de l'expérience offerte par les appareils mobiles.*

*Une PWA se consulte comme un site web classique, depuis une URL sécurisée mais permet une **expérience utilisateur similaire à celle d'une application mobile, sans les contraintes de cette dernière (soumission aux App-Stores, utilisation importante de la mémoire de l'appareil...).***

Elles proposent de conjuguer rapidité, fluidité et légèreté tout en permettant de limiter considérablement les coûts de développement.

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

Manifeste

*Le **manifeste** d'une application web fournit des informations concernant celle-ci (comme son nom, son auteur, une icône et une description) dans un document texte JSON. **Le but du manifeste est d'installer des applications sur l'écran d'accueil d'un appareil, offrant aux utilisateurs un accès plus rapide et une expérience plus riche.***

```
{
  "name": "HackerWeb",
  "short_name": "HackerWeb",
  "start_url": ".",
  "display": "standalone",
  "background_color": "#fff",
  "description": "A readable Hacker News app.",
  "icons": [{
    "src": "images/touch/homescreen48.png",
    "sizes": "48x48",
    "type": "image/png"
  }, ...],
}
```

Service Workers

*Les **service workers** jouent essentiellement le rôle de **cache** entre une application web, et le navigateur ou le réseau. Ils sont destinés à permettre la création d'expériences de navigation déconnectées efficaces, en interceptant les requêtes réseau et en effectuant des actions appropriées selon que le réseau est disponible et que des ressources mises à jour sont à disposition sur le serveur. Ils permettent aussi d'accéder aux APIs de **notifications du serveur** (push) et de **synchronisation en arrière-plan**.*

Un service worker est un worker enregistré auprès d'une origine et d'un chemin. Il prend la forme d'un fichier JavaScript.

