Architectures Web et Mobile

Sommaire

1. **Architectures logicielles**
   - 1.1. **Définition**
   - 1.2. **Principes fondamentaux**
   - 1.3. **Modèles d'architectures logicielles**
2. **Architectures en couches**
   - 2.1. **Définitions**
   - 2.2. **Types d'architectures**
   - 2.3. **Architectures *n*-couches**
3. **Architecture des composants**
   - 3.1. **Définitions**
   - 3.2. **Architecture des bases de données**
   - 3.3. **Architecture des interfaces utilisateur**
4. **Patrons de conception**
   - 4.1. **MVC : Modèle-Vue-Contrôleur**
   - 4.2. **MVP : Modèle-Vue-Présenteur**
   - 4.3. **MVVM : Modèle-Vue-Vue Modèle**

1. Architectures logicielles

1. **Définitions**
2. **Principes fondamentaux**
3. **Modèles d'architectures logicielles**

1.1. Définitions

- *Architecture* : Disposition, ordonnance d'un édifice. *INFORM.* Structure générale inhérente à un système informatique.

Une architecture logicielle est une représentation abstraite d'un système informatique. Elle consiste à :

- Décrire l'**organisation** d'un système informatique et sa décomposition en sous-systèmes ou en composants logiciels;
- Déterminer les **interfaces** qui relient les sous-systèmes ou les composants logiciels;
- Décrire les **interactions** entre les sous-systèmes ou les composants logiciels;
- Décrire les **implémentations** des sous-systèmes ou des composants logiciels.

La définition d'une architecture logicielle peut être effectuée à divers degrés d'abstraction. On peut alors parler de :

- **Conception générale** qui s'intéresse aux sous-systèmes d'un système informatique;
- **Conception détaillée** qui s'intéresse aux composants d'un sous-système d'un système informatique.

1.2. Principes fondamentaux

La définition d'une architecture logicielle doit répondre à plusieurs principes fondamentaux. Ceux-ci sont :

- De tenir compte des **exigences des différentes parties-prenantes** (en anglais : « *stakeholder* ») : l'architecture d'un système doit tenir compte des préoccupations des différents acteurs (propriétaire du système, utilisateur, administrateur, développeur, ...) concernés par le système.
- La **séparation des préoccupations** : Pour réduire la complexité d'une architecture logicielle, on peut décrire une architecture logicielle selon le point de vue de chacune des parties prenantes. On parle alors de *vues architecturales*.
- La prise en compte d'**exigences de qualité logicielle** : On peut citer la **résilience aux pannes**, la **rétro-compatibilité**, l'**extensibilité**, la **stabilité**, la **maintenabilité**, la **disponibilité**, la **sécurité**, l'**utilisabilité**, ...
- La **réutilisation de modèles** d'architecture existants : Certains modèles d'architectures qui ont fait leurs preuves et sont considérés comme des standards (architecture client-serveur, architecture REST, Architecture Orientée Services - SOA, ...).

1.3. Modèles d'architectures logicielles

Dans le cadre de l'étude des architectures Web et mobile, nous nous intéresserons particulièrement aux modèles d'**architectures physiques** et aux modèles d'**architectures de développement** des systèmes.

L'**architecture physique** ou architecture de déploiement d'un système décrit la topologie des sous-systèmes ou composants logiciels sur la couche physique du système ainsi que les connexions physiques entres les différents sous-systèmes ou composants logiciels. Nous nous intéresserons, dans ce cadre, aux architectures **distribuées** ou **centralisées** en **couches**.

L'**architecture de développement** s'attache à décrire l'implémentation d'un sous-système ou d'un composant logiciel du point du programmeur. Nous nous intéresserons, dans ce cadre, à différents patrons de conception (en anglais, « *design pattern* ») tels que le MVC (Modèle Vue Contrôleur), le MVP (Modèle Vue Présenteur), le MVVM (Modèle, Vue, Vue Modèle).

2. Architectures en couches

1. **Définitions**
2. **Types d'architectures**
3. **Architectures *n*-couches**

2.1. Définitions

- Couche (en anglais « tier », *trad.* niveau) : *INFORM.* Subdivision de l'architecture physique ou logique d'un système informatique.

Une application web ou une application pour terminal mobile est, avant toute chose, un logiciel qui s'appuie sur une architecture physique

Les architectures physiques peuvent être à 1, 2 ,3 ou n couches (en anglais « tier », *trad.* niveau). Les couches sont matérialisées par la séparation physique des logiques inhérentes aux développements logiciels; chaque couche étant gérée par un système informatique distinct.

On peut généralement distinguer les couches par leur logique :

- la logique de présentation (concerne l'affichage),
- la logique de traitement (concerne les accès, les opérations …)
- la logique de données (concerne le stockage, la manipulation des données, etc…).

Certaines couches peuvent elles-mêmes faire l'objet de subdivisions, on parle alors de n (n, inconnue) couches.

Architecture multi-couches **En anglais, « *Multi-tier architecture* »**	
Système Informatique A	Couche de présentation, Interface utilisateur (en anglais « *Presentation tier* »)
Système Informatique B	Couche de traitement, métier (en anglais « *Logic tier (or business)* »)
Système Informatique C	Couche de données (en anglais « *Data tier* »)

Les architectures à 1 ou plusieurs couches sont généralement construites à l'aide de « *clients* » et de « *serveurs* ».

Notion de [logiciel] client :

- *Client* : *INFORM.* Équipement informatique sur lequel est installé un *logiciel client*. (ex : un ordinateur, un smartphone, une console de jeux vidéos, une smart TV, …).
- *Logiciel client* : *INFORM.* Logiciel qui permet à un équipement informatique d'établir une communication avec un autre équipement informatique. (ex : un logiciel de discussion instantanée, un logiciel de partage de fichiers Peer2Peer, un logiciel de messagerie, …)

Un « *client* » est un équipement informatique sur lequel est installé un logiciel client. Les logiciels « *clients* » peuvent établir des communications avec des logiciels « *serveurs* » à travers un réseau. Dans le cas des applications Web ou mobiles, on distinguera 2 types de clients :

- Les clients lourds (ou riches ; en anglais « *thick/fat clients* » ou « *rich clients* ») qui prennent généralement en charge la logique d'affichage, de traitement et parfois même de données. L'application Clash of Clans©, par exemple, sur un smartphone ou une tablette sous iOS© ou Android© est plutôt un client lourd.
- Les clients légers (en anglais « *thin clients* ») qui prennent généralement en charge uniquement la logique d'affichage. L'application FaceBook®, par exemple, sur un smartphone ou une tablette sous iOS© ou Android© est plutôt un client léger.

On considère que les applications Web (accessibles uniquement à l'aide du navigateur Web) ou les applications qui sont simplement des conteneurs pour un site Internet mobile (« *Web App* ») sont

des clients légers alors que les applications qui nécessitent beaucoup plus de traitements sur le terminal mobile sont des clients lourds.

Notion de [logiciel] serveur

- *Serveur* : *INFORM.* Équipement informatique sur lequel est installé un *logiciel serveur*. (ex : un ordinateur, un smartphone, une console de jeux vidéos, une smart TV, …).
- *Logiciel serveur* : *INFORM.* Logiciel qui permet à un équipement informatique de recevoir et de répondre à une communication établie par un autre équipement informatique.

Un « *serveur* » est un équipement informatique sur lequel est installé un logiciel serveur. Supposons que le logiciel serveur Apache soit installé sur mon téléphone portable, mon téléphone portable peut alors être qualifié de « *serveur* ».

Lorsqu'un logiciel serveur est installé sur un ordinateur équipé du système d'exploitation Linux (ou une de ses variantes), on dit communément « *serveur Linux* » (cela décrit un ordinateur sur lequel est installé le système d'exploitation Linux ainsi qu'un logiciel serveur). Il en va de même pour un « *serveur Windows* » ou un « *serveur Mac OS* » ou un « *serveur Solaris* » ou un « *serveur Unix* » ou …

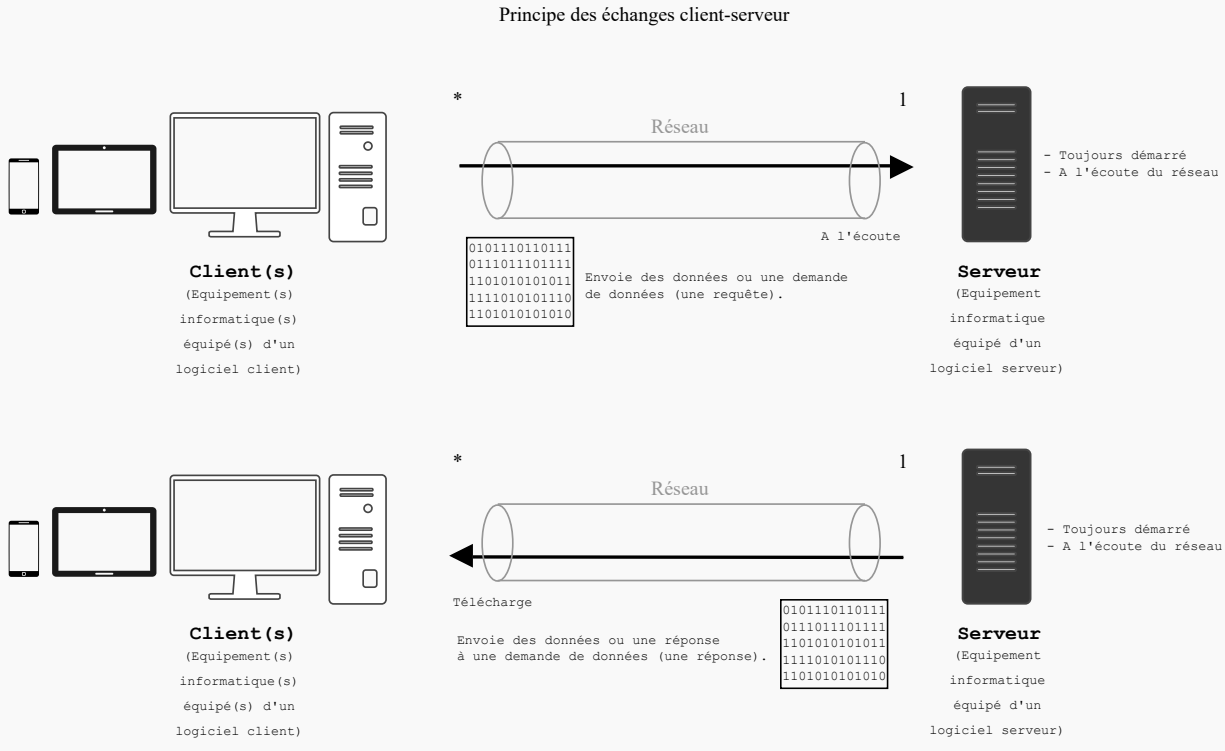
« *Serveur* »	
En anglais, « *Server* »	
Logiciel Serveur	*(Apache, Nginx, Lighttpd, MySQL, PostgreSQL SQL, Oracle©, ProFTPd, SSHd, Samba …)*
Système d'exploitation	*(Linux, Windows©, Mac OS©, Unix, Solaris …)*
Équipement informatique	

Un logiciel serveur est un logiciel qui a pour mission de recevoir et de répondre à des communications établies par d'autres équipements informatiques selon les règles de certains **protocoles de communication** associés aux réseaux informatiques.

Pour que ces logiciels puissent remplir leur mission, ils doivent être tout le temps démarrés, en « *tâche de fond* », puisqu'ils sont censés être à l'« *écoute du réseau* » à tout instant. On les qualifie alors, lorsqu'ils sont démarrés, de **services** ou de **daemons**.

Notion d'échanges client-serveur :

Les équipements informatiques sur lesquels sont installés des logiciels serveurs reçoivent et répondent aux communications établies par des équipements informatiques sur lesquels sont installés des logiciels clients. On peut alors parler d'échanges « ***client-serveur*** » :



Il existe de nombreuses familles de logiciels serveurs et chacune d'entre elles est associée à une utilité particulière. Dans le cas des applications Web ou mobiles, un ou plusieurs serveurs peuvent prendre en charge une partie ou la totalité de la logique de traitement et/ou de données.

Un logiciel client et un logiciel serveur peuvent être installés sur la même machine. On parle dans ce cas de serveur local. Le logiciel client utilise alors l'interface réseau locale (c'est à dire le réseau « interne » de la machine) pour se connecter au serveur local.

2.2. Types d'architectures

Les architectures à 2 ou plusieurs couches peuvent être **centralisées** (en anglais « *centralized* ») ou **distribuées** (en anglais « *distributed* »).

Notion d'architecture centralisée :

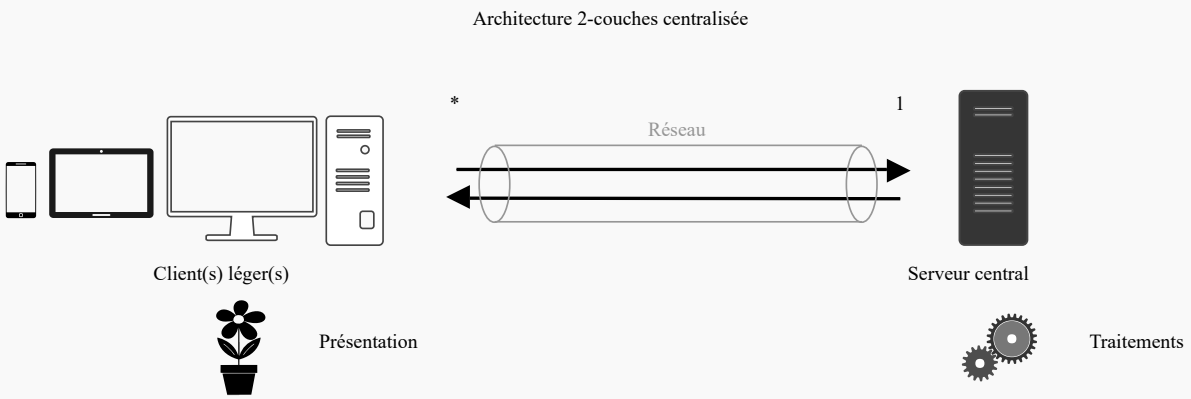
On parle d'architecture **centralisée** lorsque les traitements sont effectués sur un serveur central. Le clients, quant à eux, ne servent qu'à la présentation et à la saisie des données. Ces clients sont alors considérés comme faisant partie des **clients légers**.

Généralement, un client léger ne peut pas fonctionner si le terminal est hors-ligne. Par exemple, l'application Twitter© sur iOS© ou Android© ne fonctionne pas si le téléphone n'est pas connecté au réseau Internet.

Les applications Web (en anglais, « *Web application* » ou « *Web App* ») peuvent être considérées, la plupart du temps, comme des clients légers. Elles prennent en charge toute la logique d'affichage. La logique de traitement et la logique de stockage des données sont pris en charge par des machines distantes accessibles via le réseau (Internet généralement). Les Applications Web se présentent :

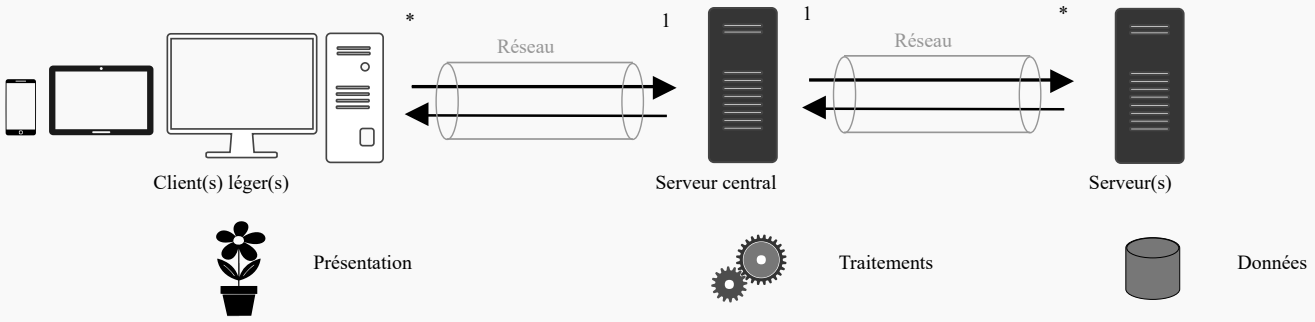
- soit sous la forme d'une application mobile à l'intérieur de laquelle on utilise un conteneur Web (« *Web container* » ou « *Web view* ») qui est une sorte de navigateur Internet léger incorporé dans l'application;
- soit sont utilisables via le navigateur Web. Pour assurer la compatibilité avec les terminaux mobile, on fait en sorte que la présentation de l'application Web soit adaptative (c.f. « *Responsive Web Design* » ou « *Adaptative Web Design* »).

Les applications Web ou mobile **centralisées** peuvent être réparties sur 2 couches :



Mais le plus communément, elle sont réparties sur **3 couches** :

Architecture 3-couches centralisée



L'**architecture client-serveur centralisée sur 3 couches** est à l'heure actuelle l'architecture la plus commune pour les applications Web.

Ce type d'architecture présente les avantages et inconvénients suivants :

- **Avantages :**
  - La sécurité : Les traitements sont centralisés et, par conséquent, le contrôle des traitements est plus simple à mettre en œuvre.
  - La stabilité : Les traitements ainsi que les données sont centralisés et, par conséquent, si le terminal client venait à être indisponible, l'utilisation d'un autre terminal permet généralement à l'utilisateur de retrouver l'application dans l'état dans lequel elle était avant l'indisponibilité.
  - L'extensibilité : Le déploiement de nouvelles fonctionnalités ne nécessite généralement que d'intervenir au niveau du serveur central.
  - La rétro-compatibilité : De la même façon, la gestion de la rétro-compatibilité ne nécessite généralement que d'intervenir au niveau du serveur central.
- **Inconvénients :**
  - L'utilisabilité : Les performances d'une application dont l'architecture est centralisée dépendent fortement de la qualité du réseau entre le client et le serveur central.
  - La disponibilité : De la même façon, en cas de coupure de réseau et donc de rupture de communication avec le serveur central, l'application devient indisponible.
  - La résilience aux pannes : Enfin, toute sorte d'aléa qui pourrait avoir un impact sur le bon fonctionnement du serveur central entraînerait une indisponibilité générale de l'application.

Plus généralement, une des critiques récurrente des architectures centralisées concerne la **sous-exploitation des capacités de traitement** disponibles sur les terminaux client.

Notion d'architecture distribuée :

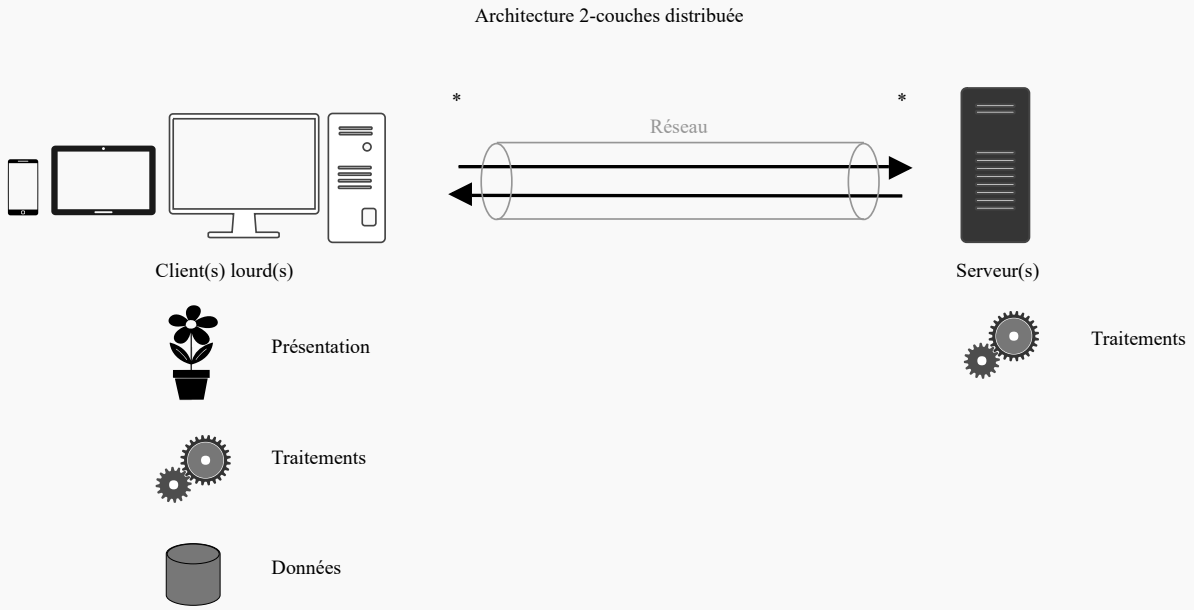
On parle d'architecture **distribuée** lorsque les traitements sont effectués en partie sur le client et en partie sur un ou plusieurs serveurs. Ces clients sont alors considérés comme faisant partie des **client lourds**.

Généralement, un client lourd peut fonctionner même si le terminal est hors-ligne. Par exemple, l'application Asphalt 8© sur iOS©/Android© fonctionne même si le téléphone n'est pas connecté au réseau Internet avec certaines fonctionnalités de jeu en moins (jeu multijoueur, évènements, nouveaux véhicules, etc…).

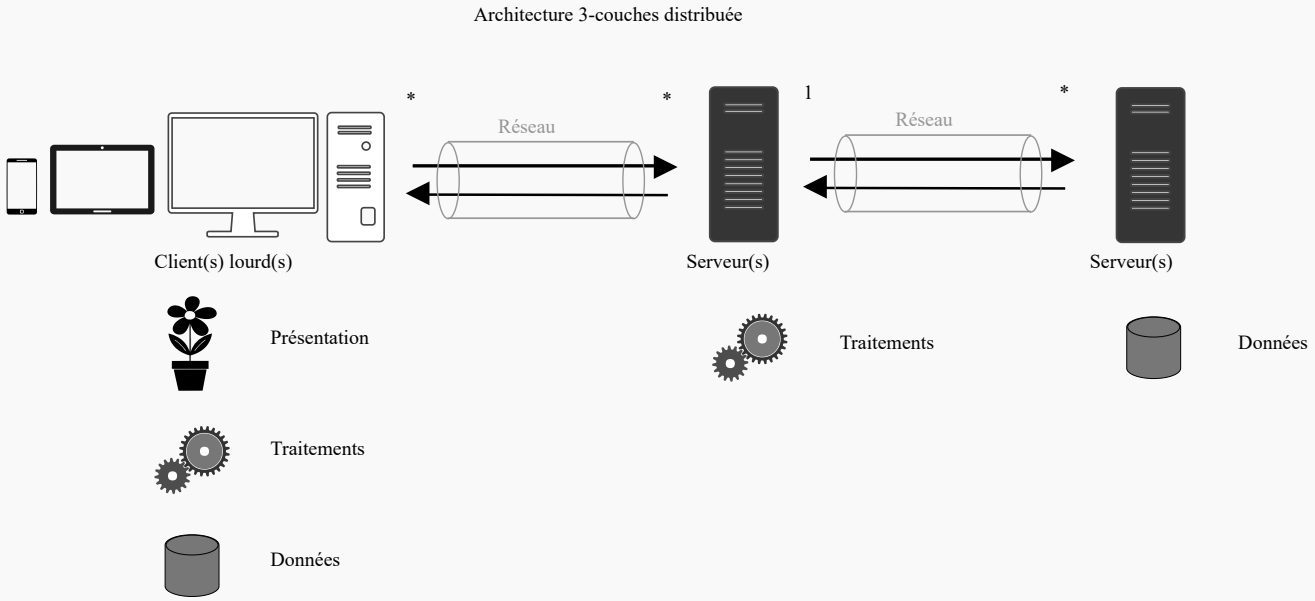
Les clients lourds sont des applications web ou mobile qui prennent en charge l'affichage mais aussi tout ou une partie de la logique de traitement et de stockage des données. Ils peuvent, dans certains cas, fonctionner tout aussi bien en étant hors-ligne (déconnecté du réseau Internet) qu'en étant en-ligne.

Lorsque l'application est hors-ligne, elle stocke ses données localement, c'est-à-dire en utilisant le système de gestion de bases de données du terminal. Et, lorsqu'elle est en ligne, elle synchronise tout ou une partie des données locales avec un système de gestion de base de données distant, c'est-à-dire accessible via le réseau Internet.

Les applications Web ou mobile **distribuées** peuvent être réparties sur 2 couches :



Mais le plus communément, elle sont réparties sur **3 couches** :



L'**architecture client-serveur distribuée sur 3 couches** est à l'heure actuelle l'architecture la plus commune pour les applications mobiles.

Ce type d'architecture présente les avantages et inconvénients suivants :

- **Avantages :**
    - L'utilisabilité : Les performances de l'application dépendent en partie de la qualité du réseau et en partie des performances du terminal client.
    - La disponibilité : L'application reste disponible même en cas de coupure de réseau et donc de rupture de communication avec le serveur.
    - La résilience aux pannes : Une panne du serveur n'entraîne pas forcément une indisponibilité de l'application, de même qu'une panne de l'application sur un terminal client n'entraîne pas une défaillance générale du système sur tous les terminaux client.

- **Inconvénients :**
    - La sécurité : Les traitements ne sont pas centralisés et, par conséquent, le contrôle des traitements est plus difficile à mettre en œuvre.
    - La stabilité : Une perte de données sur le terminal client ou une défaillance du terminal client ne permet généralement pas à l'utilisateur de retrouver l'application dans l'état dans lequel elle était après la résolution du problème.
    - L'extensibilité : Le déploiement de nouvelles fonctionnalités nécessite d'intervenir au niveau du serveur ainsi qu'au niveau de l'ensemble des terminaux sur lesquels est installé le client lourd.
    - La rétro-compatibilité : De la même façon la gestion de la rétro-compatibilité nécessite généralement d'intervenir au niveau du serveur mais aussi au niveau de l'ensemble de terminaux sur lesquels est installé le client lourd.

Plus généralement, une des critiques récurrentes des architectures centralisée concerne la **sur-exploitation des capacités de traitement** disponibles sur les terminaux client, en particulier des terminaux mobiles.

2.3. Architectures *n*-couches

La multiplication des couches permet de répondre à plusieurs problématiques.

On peut optimiser les performances en libérant de la charge de travail pour les traitements ou en libérant de la bande passante. On peut utiliser des serveurs de données « *statiques* » pour héberger les contenus « *statiques* » comme les images, les vidéos, les fichiers texte, …

On peut aussi multiplier les sources de données pour la réalisation des traitements et procéder à l'agrégation des données provenant d'autres serveurs applicatifs qui mettent à disposition des contenus formatés selon des **formats d'échange** de données normalisés.

Notion de format d'échange de données :

- Format d'échange : Il s'agit d'un format de données dont on est sûr que le ou les destinataires disposent de l'outil qui permet de le visualiser.
- XML : (eXtensible Markup Language, « *langage de balisage extensible* ») est un langage informatique de balisage qui permet de décrire une arborescence de données.
- JSON (JavaScript Object Notation) : Il s'agit est un format de données textuel, générique, dérivé de la notation des objets du langage ECMAScript.

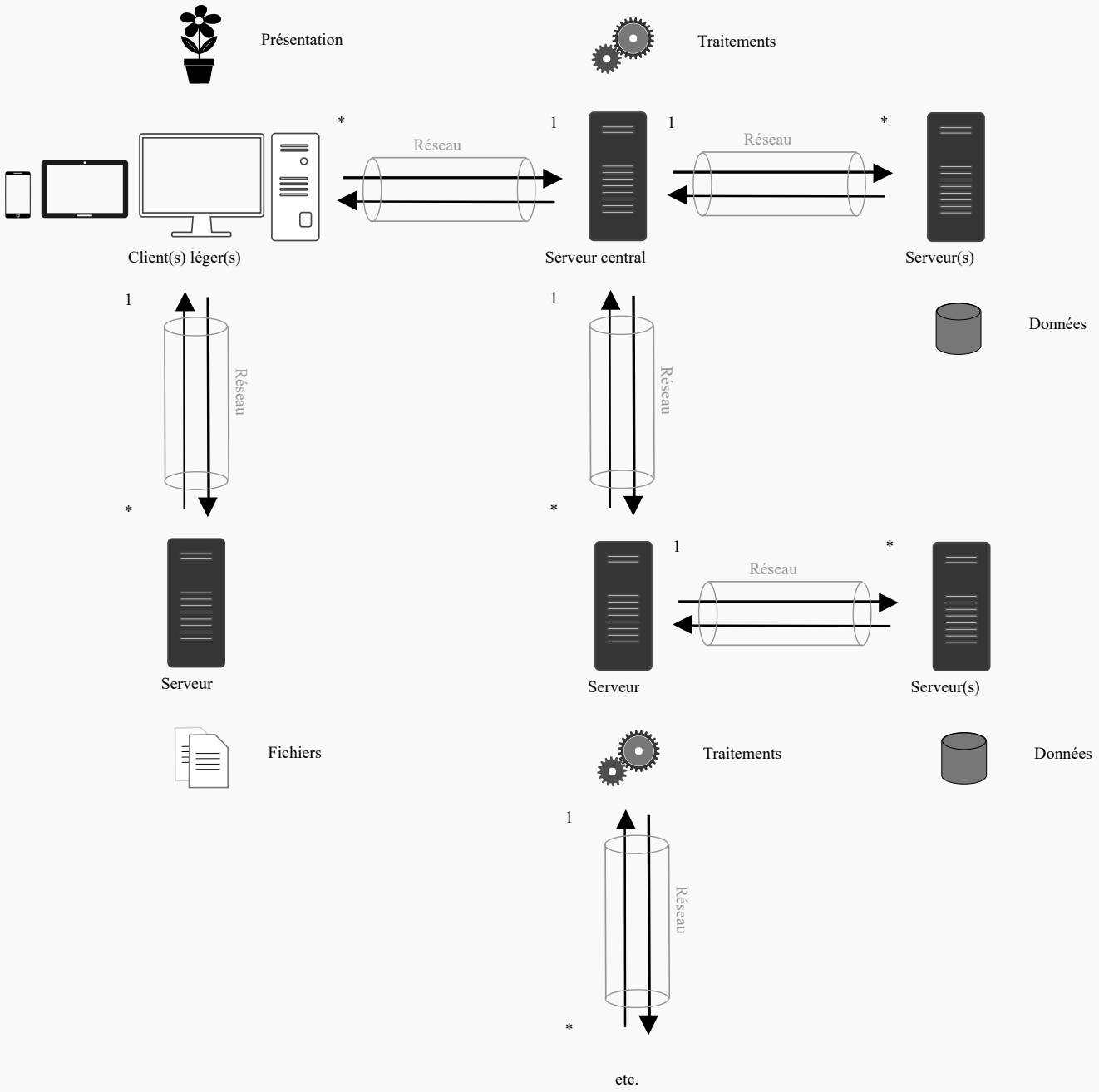
Le XML ainsi que le JSON (JavaScript Object Notation) sont les formats d'échange de données les plus utilisés pour les applications Web.

Notion de Service Web

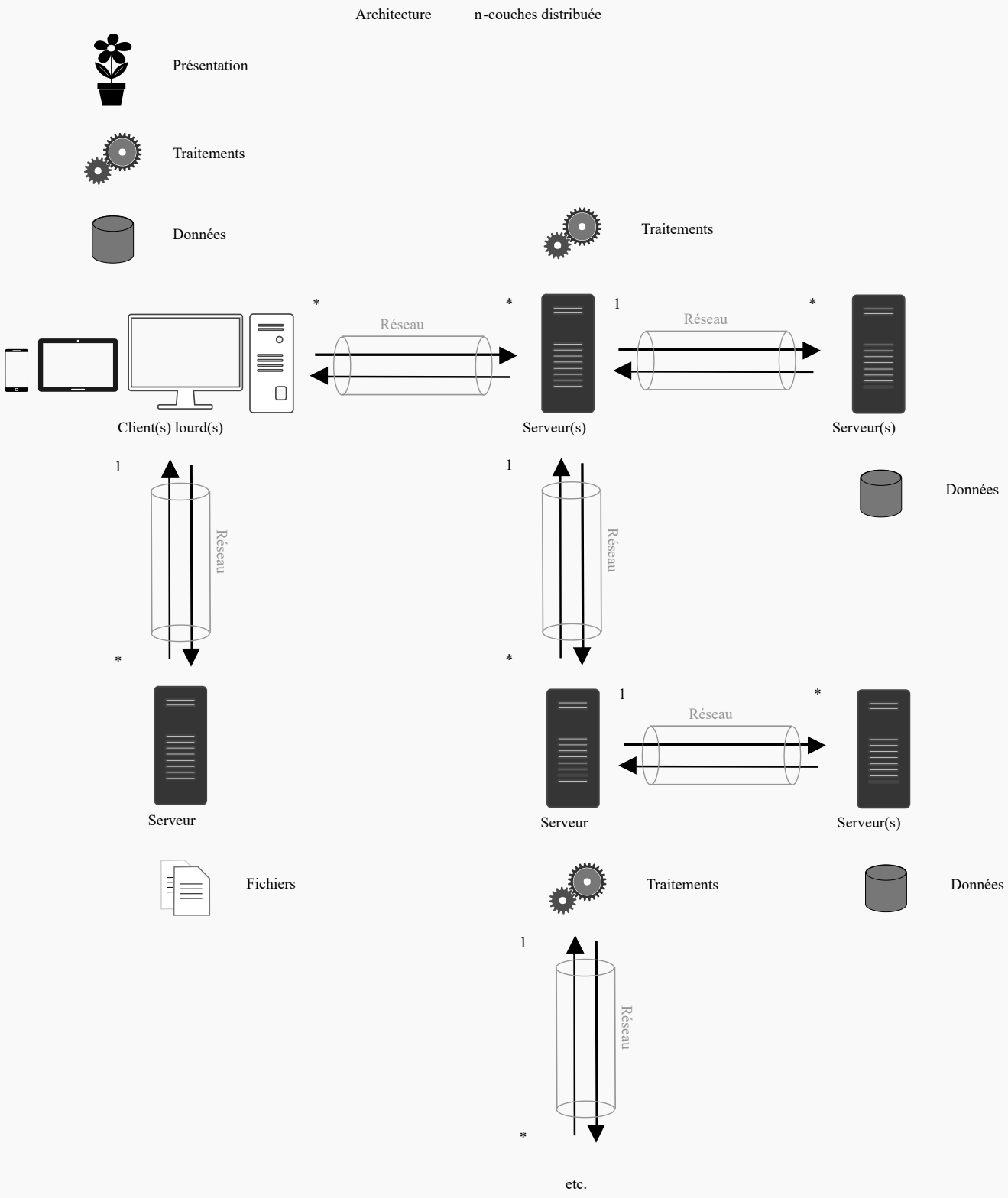
- Service Web : Ensemble de fonctionnalités accessibles via Internet ou un intranet, par et pour des applications ou machines, sans intervention humaine, et de manière synchrone
- WSDL : Grammaire XML permettant de décrire un Service Web basé sur le format d'échange XML.
- API REST : Documentation permettant de décrire un Service Web basé sur des URI et le format d'échange JSON.

Les Services Web sont essentiellement utilisés pour permettre à des serveurs applicatifs de communiquer entre eux pour s'échanger des données selon les formats d'échange pour le Web.

On pourrait décrire une architecture *n*-couches **centralisée** pour une application Web ou une application mobile comme suit :



Et une architecture *n*-couches **distribuée** pour une application Web ou une application mobile comme suit :



La conception et le développement d'une application mobile peuvent donc faire l'objet de réalisations réparties sur une ou plusieurs couches. Sur chacune de ces couches, les programmes devront être réalisés en utilisant un ensemble de méthodes de développement qui vont de la modélisation de l'architecture des programmes aux techniques de programmation modernes.

3. Architecture des composants

1. **Définitions**
2. **Architecture des bases de données**
3. **Architecture des interfaces utilisateur**

3.1. Définitions

Un sous-système ou un composant logiciel peut faire l'objet d'une **conception détaillée**. On parle alors d'**architecture de développement** ou d'architecture logique.

Lorsque les architectures de développement ou architectures logiques concernent :

- la **couche de stockage** des données on parle d'**architecture de base de données**;
- la **couche de traitement et de présentation**, on parle d'**architecture des interfaces utilisateur**.

Ces architectures s'appuient sur des **techniques de modélisation ou d'implémentation logicielle** qui sont généralement **indépendantes des langages** de programmation utilisé.

3.2. Architecture des bases de données

Les bases de données peuvent être **hiérarchiques** (basées généralement sur les protocoles X.500 comme le protocole LDAP), relationnelles (basées généralement sur le langage SQL) ou **non relationnelles - *NoSQL*** (plutôt orientées objet-relationnel et basées généralement sur le langage ECMAScript).

Les méthodes de modélisation d'architecture pour :

- les **bases de données hiérarchiques** sont :
  - Une représentation hiérarchique respectant les spécifications du **protocole X.500** concernant les **DIT** (en anglais « *Directory Information Tree* »).
- les **bases de données relationnelles** sont :
  - La méthode de **Modélisation Conceptuelle de Données** (MCD - en anglais « *CDM - Conceptual data model* ») Merise; puis,
  - La méthode de **Modélisation Logique de Données** (MLD - en anglais « *LDM - Logical data model* ») de base de données UML (en anglais « *Unified Modeling Language* »).
- les **bases de données non relationnelles** sont :
  - Les **méthodes de modélisation pour les bases de données relationnelles** dans le cas d'une approche relationnelle; ou,
  - La méthode de **Modélisation Objet** UML dans le cas d'une approche objet.

Les méthodes de modélisation de bases de données font partie des cours sur les Systèmes de Gestion de Bases de Données (SGBD), c'est pourquoi nous n'aborderons pas plus en détail ce sujet dans ce document.

3.3. Architecture des interfaces utilisateur

Par interfaces utilisateur, on entend couche de présentation et/ou couche de traitement. Les architectures des interfaces utilisateur s'appuient sur des patrons de conception (en anglais, « *Design Patterns* ») qui définissent l'articulation générale des différents composants logiciels d'un sous-système. Les principaux patrons de conception qui peuvent être mis en oeuvre sont :

- Le patron de conception **Modèle-Vue-Contrôleur** (MVC - en anglais « *MVC - Model-View-Controller* ») lancé en 1978,
- Le patron de conception **Modèle-Vue-Présenteur** (MVP - en anglais « *MVP - Model-View-Presenter* ») lancé en 1990,

- Le patron de conception **Modèle-Vue-Vue Modèle** (MVVM - en anglais « *MVVM - Model-View-ViewModel* ») lancé en 2005.

Ces différents patrons de conception ont pour objectifs de :

- Respecter des « *bonnes pratiques* »,
- Minimiser les coûts et les besoins de maintenance,
- Favoriser l'utilisabilité et l'extensibilité,
- Faciliter la compréhensibilité.

Pour atteindre ces objectifs, ils reposent tous sur les principes suivants :

- La **séparation des préoccupation** : diviser l'application en parties distinctes dont les fonctionnalités se chevauchent le moins possible pour favoriser la compréhensibilité,
- La **responsabilité unique** : chaque sous-composant de l'application est responsable d'une seule et unique fonctionnalité ou caractéristique pour favoriser la réutilisabilité,
- La **minimisation de l'effort initial** : l'initialisation du patron de conception ne doit pas nécessiter un gros effort d'implémentation pour favoriser l'extensibilité,
- La **suppression des répétition** : les fonctionnalités ne sont pas dupliquées au sein d'une application pour favoriser la maintenabilité,
- Dans le cas des langages orienté objet, la **composition plutôt que l'héritage** : l'utilisation de la composition plutôt que de l'héritage lorsqu'on réutilise une fonctionnalité pour favoriser la flexibilité,
- ... etc.

Ces patrons de conceptions ne sont pas forcément liés à des approches de la programmation telle que la programmation orienté objet. Il peuvent être mis en oeuvre à l'aide de langages de programmation purement procéduraux. Il n'y a **pas de lien entre programmation orienté objet et patrons de conception** mais l'utilisation d'un patron de conception dans le cadre d'un programme basé sur une approche orienté objet apporte des bénéfices non négligeables à son implémentation.

4. Patrons de conception

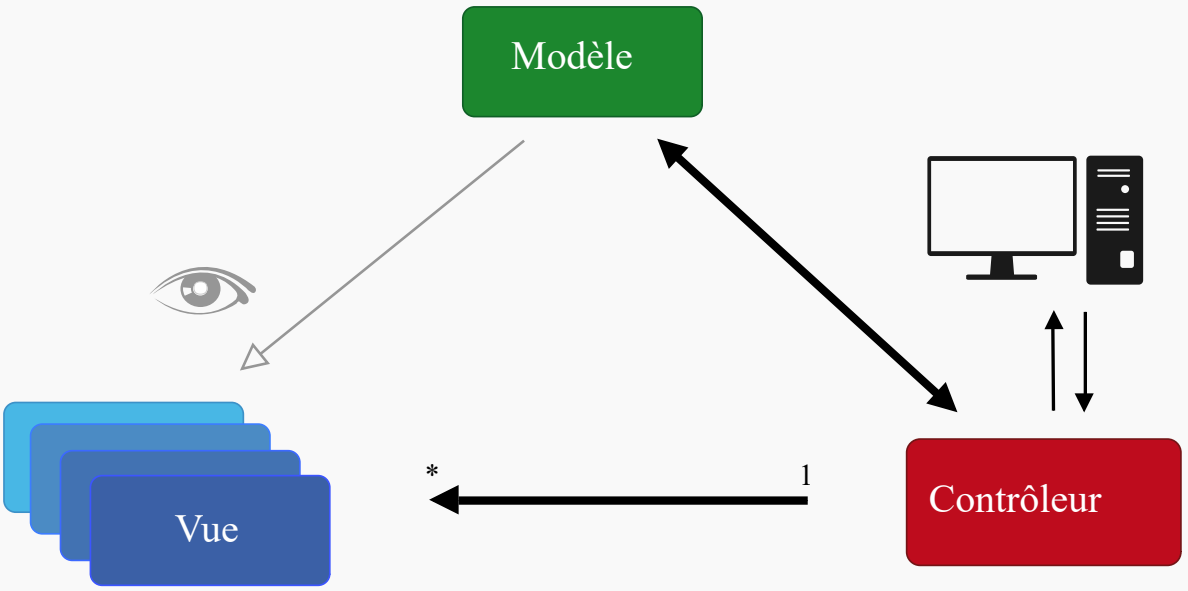
1. **MVC : Modèle-Vue-Contrôleur**
2. **MVP : Modèle-Vue-Présenteur**
3. **MVVM : Modèle-Vue-Vue Modèle**

4.1. MVC : Modèle-Vue-Contrôleur

Le patron de conception MVC est très populaire pour le développement d'application Web centralisées, en particulier pour la couche de traitement et de présentation.

Le MVC propose de subdiviser le composant logiciel en couches qui sont :

- Le **Modèle** :
  - Peut lire et écrire dans la base de données;
  - Peut contenir des données de l'application;
  - Peut contenir des fonctions ou méthodes pour manipuler ces données;
- Le **Contrôleur** :
  - Peut contenir des **Actions**;
  - Transmets des données à afficher aux Vues;
  - Peut utiliser un Modèle pour accéder aux données;
  - Est un intermédiaire entre plusieurs Vue et Modèles;
  - Observe les changements des Modèles et, le cas échéant, les transmet aux Vues concernées;
  - Vérifie l'intégrité technique et fonctionnelle des données fournies par les utilisateurs.
- La **Vue** :
  - Gère les interactions avec l'utilisateur
  - Est une représentation visuelle d'un Modèle



- Une Vue n'a pas de visibilité sur le Contrôleur ou l'Action qui l'a appelée;
- Une Vue observe les changements des données d'un Modèle et s'y adapte en conséquence;

- Une Vue fait référence et s'attend à des données d'un Modèle en particulier ou à un Modèle en particulier;
- Un Contrôleur ou une Action au sein d'un Contrôleur peut appeler plusieurs Vues;
- Un Contrôleur ou une Action au sein d'un Contrôleur peut passer des données d'un Modèle ou un Modèle à une Vue;
- Un Contrôleur ou une Action au sein d'un Contrôleur est le **point d'entrée** d'une fonctionnalité de l'application.

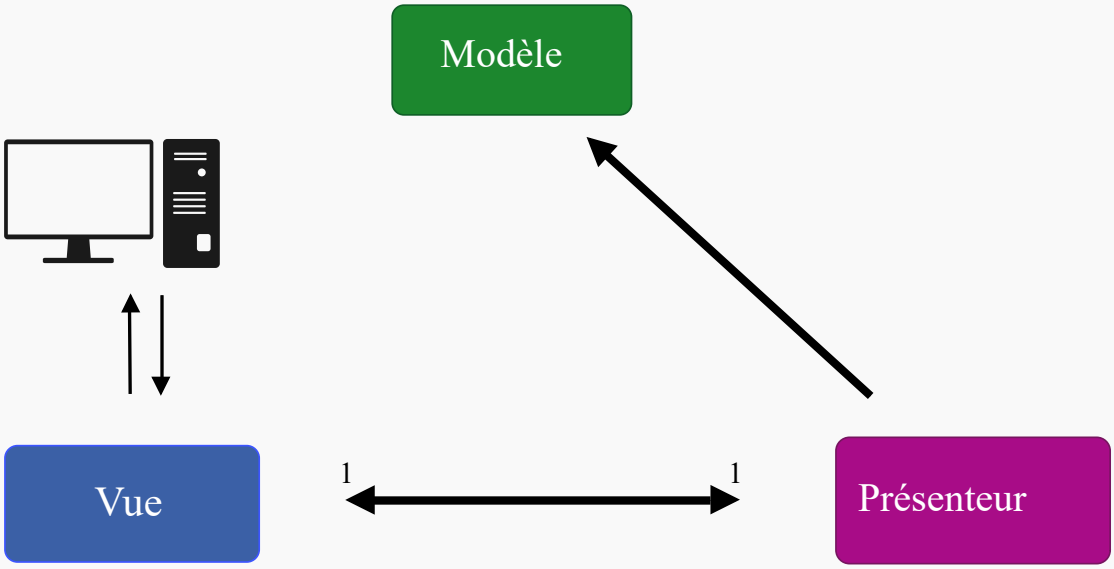
Certains « *frameworks* » (en français, cadres de travail) de développement pour applications Web très connus sont basés sur le MVC tels que « *Symfony 3* » ou « *Laravel* » en PHP, « *ExpressJS* » en JavaScript, « *Ruby On Rails* » en Ruby, ...

4.2. MVP : Modèle-Vue-Présentateur

Le patron de conception MVP est utilisé plus particulièrement pour le développement d'applications Web distribuées :

Le MVP propose de subdiviser le composant logiciel en couches qui sont :

- Le **Modèle** :
  - Peut lire et écrire dans la base de données;
  - Peut contenir des données de l'application;
  - Peut contenir des fonctions ou méthodes pour manipuler ces données;
- Le **Présenteur** :
  - Transmets des données à afficher aux Vues;
  - Peut utiliser un Modèle pour accéder aux données;
  - Est un intermédiaire entre une et unique Vue et un ou plusieurs Modèles;
  - Observe les changements des Modèles et, le cas échéant, les transmet aux Vues concernées;
  - Vérifie l'intégrité technique et fonctionnelle des données fournies par les utilisateurs.
- La **Vue** :
  - Gère les interactions avec l'utilisateur
  - Est une représentation visuelle d'un Présenteur



- Une Vue à forcément un Présenteur associé;
- Les Vues ne communiquent pas avec les Modèles;
- Une Vue invoque le Présenteur qui l'a appelée;
- Une Vue est le **point d'entrée** d'une fonctionnalité de l'application.
- Un Présenteur est une abstraction d'une Vue;
- Un Présenteur fait référence à une Vue et vice-versa;
- Le Présenteur formatte les données d'un Modèle et les transmet à une Vue;

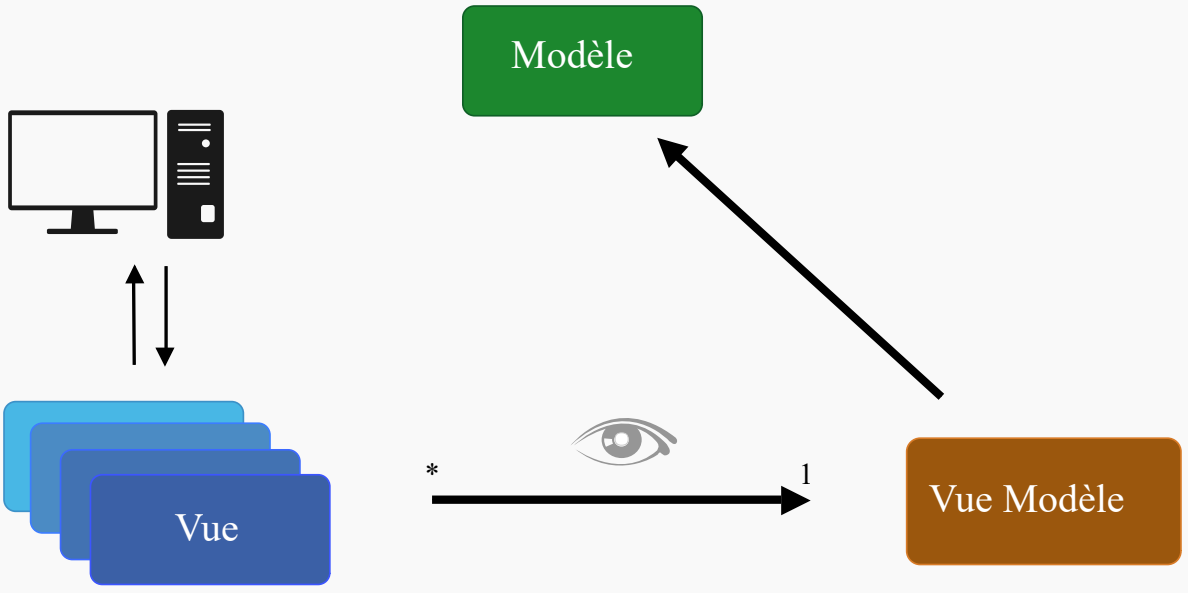
On peut citer le « *frameworks* de développement pour applications Web « *BackBone JS* » en JavaScript qui permet d'implémenter du MVP.

4.3. MVVM : Modèle-Vue-Vue Modèle

Le patron de conception MVVM est très populaire pour le développement d'applications Web distribuées :

Le MVVM propose de subdiviser le composant logiciel en couches qui sont :

- Le **Modèle** :
  - Peut lire et écrire dans la base de données;
  - Peut contenir des données de l'application;
  - Peut contenir des fonctions ou méthodes pour manipuler ces données;
- Le **Vue Modèle** :
  - Peut utiliser un Modèle pour accéder aux données;
  - Est un intermédiaire entre une ou plusieurs Vues et un ou plusieurs Modèles;
  - Observe les changements d'une Vue et, le cas échéant, les transmet aux Modèles concernés;
  - Vérifie l'intégrité technique et fonctionnelle des données fournies par les utilisateurs.
- La **Vue** :
  - Gère les interactions avec l'utilisateur



- Les Vues ne communiquent pas avec les Modèles;

- Une Vue fait référence au propriétés de la Vue Modèle associée;
- Une Vue est le **point d'entrée** d'une fonctionnalité de l'application.
- Une Vue Modèle une abstraction fortement typée de la Vue;
- Une Vue Modèle est toujours synchronisée avec la Vue associée;
- Une Vue Modèle ou un Modèle ne référence pas la Vue associée;
- Le Modèle n'a pas de visibilité sur la Vue Modèle qui l'a appelé ou sur les Vues;

Certains « *frameworks* » de développement pour application mobiles et applications Web très connus sont basés sur le MVVM tels que « *Angular 2* » ou « *Ember* » en JavaScript, ...