
Introduction

Introduction

Le développement web a beaucoup évolué ces dernières années :

- Des nouvelles méthodologies
  - Agile, Lean Startup, ...
- Des nouvelles plateformes
  - Explosion du cloud computing
- Des nouvelles technologies
  - Node.js, NoSQL, SPA, ...

Introduction

Il est important et nécessaire de comprendre ces évolutions !

Mais pour commencer ...

Qu'est ce que le développement web en 2021 ?

Historique

Historique

Au début (1990), le web était statique.

Les serveurs servaient des pages informatives, le contenu était fixé une fois pour toute.

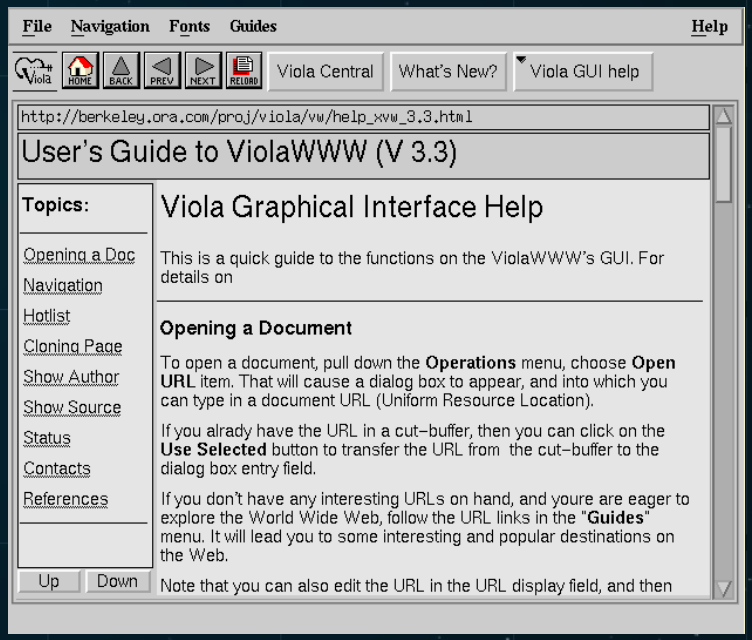
Le navigateur était chargé de faire le rendu de ces pages : très peu d'interactions avec l'utilisateur.

Historique

1991 / 1992

HTML 1.0 : « langage » permettant de structurer le contenu d'une page web

ViolaWWW : un des premiers navigateurs



26 sites web dans le monde !

Historique

1993 / 1994

De nouveaux navigateurs : **Mosaic**, **Netscape**

Apparition des images dans les pages web !

10 022 sites web fin 1994

Historique

A partir de 1995, le Web commence à intéresser les médias et le grand public.

Et surtout, c'est l'apparition de **PHP** !

Cette fois, le serveur va exécuter des scripts en réponse aux requêtes des utilisateurs : le contenu de la page est généré à la demande.

Historique

Personnalisation du contenu en fonction de l'utilisateur : c'est le début du web dynamique.

Le navigateur reste encore très basique : les interactions se limitent à cliquer sur des liens hypertextes ou des boutons.

JavaScript est créé en 1995, mais pour le moment ses capacités sont très limitées.

Historique

1996 / 1997

Flash

Applets Java

Plus de 1 000 000 de sites fin 1997

Historique

Les **applets Java** et les **applications Flash** ont permis d'amener du dynamisme sur le navigateur.

Interfaces utilisateurs plus riches, plus d'actions.

L'applet est "servie" par le serveur mais s'exécute localement dans le navigateur.

L'ancêtre des "**Single Page Applications**".

Historique

Et puis...

1998 : Moteur de recherche **Google**

2000 : **AJAX**

2004 : Concept de “**web 2.0**”, Facebook

2008 : Google **Chrome**

Aujourd’hui

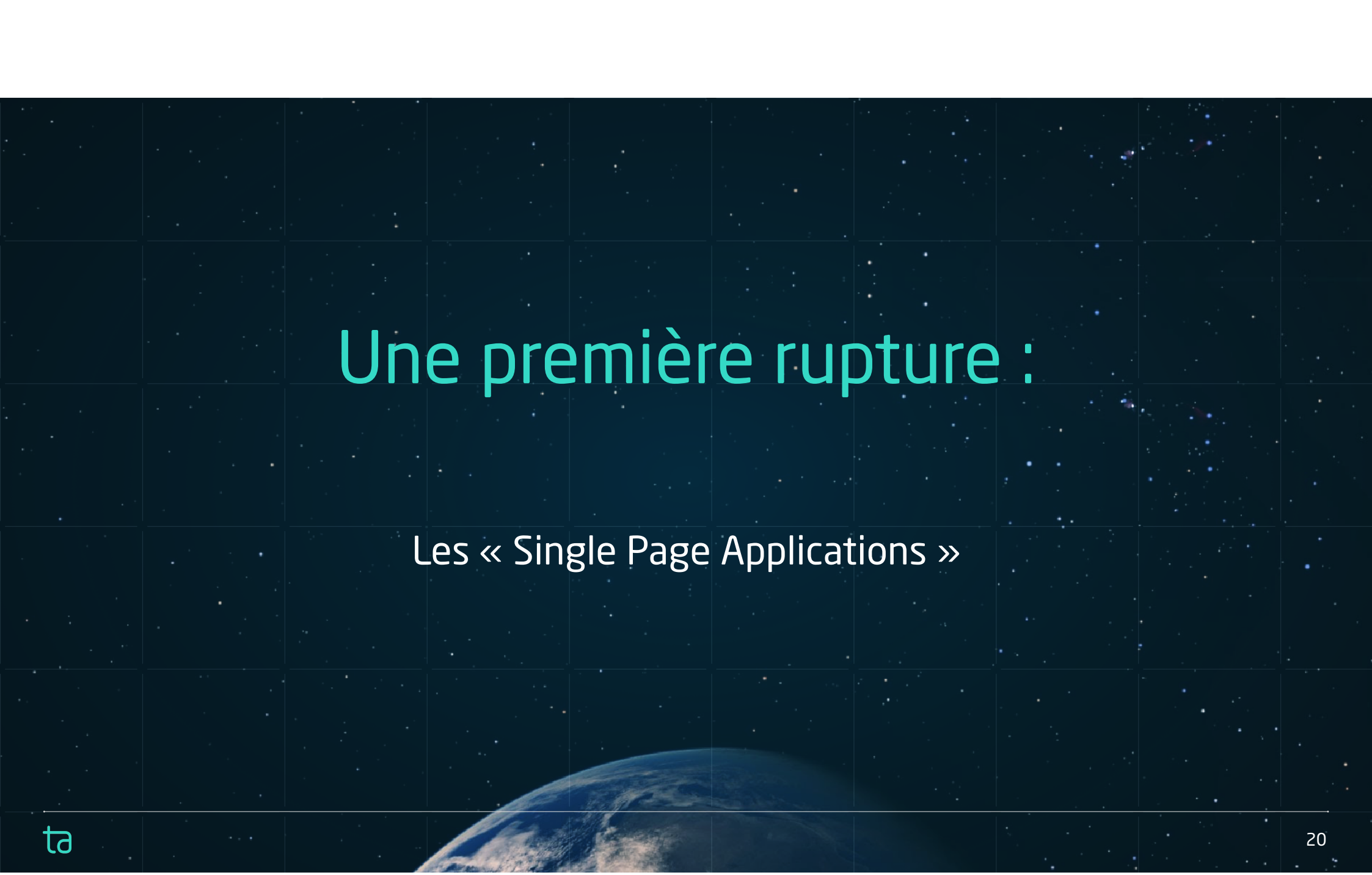
Aujourd'hui

Le navigateur est devenu aujourd'hui une plateforme d'exécution à part entière :

- **JavaScript** performant
- Chargement de données asynchrones (**AJAX**)
- Sécurité (**SSL**)
- Rendu graphique très évolué (**CSS3**, **SVG**, **Canvas**, **WebGL**)
- **HTML 5** : API Filesystem, base de données embarquées, ...

Une première rupture :

Les « Single Page Applications »



Les architectures s'adaptent à la technologie

Changement de principe structurant dans le développement :

- A l'origine, les calculs étaient tous effectués sur le serveur (**PHP**)
- Les **applets Java** ont déporté une partie des tâches sur le client. Mais cela a été progressivement abandonné : **sandboxing contraignant**, **mauvaise compatibilité** (**mobile**)
- L'état de l'art est ensuite revenu aux traitements côtés serveurs : **Java EE**, **.NET**, ...

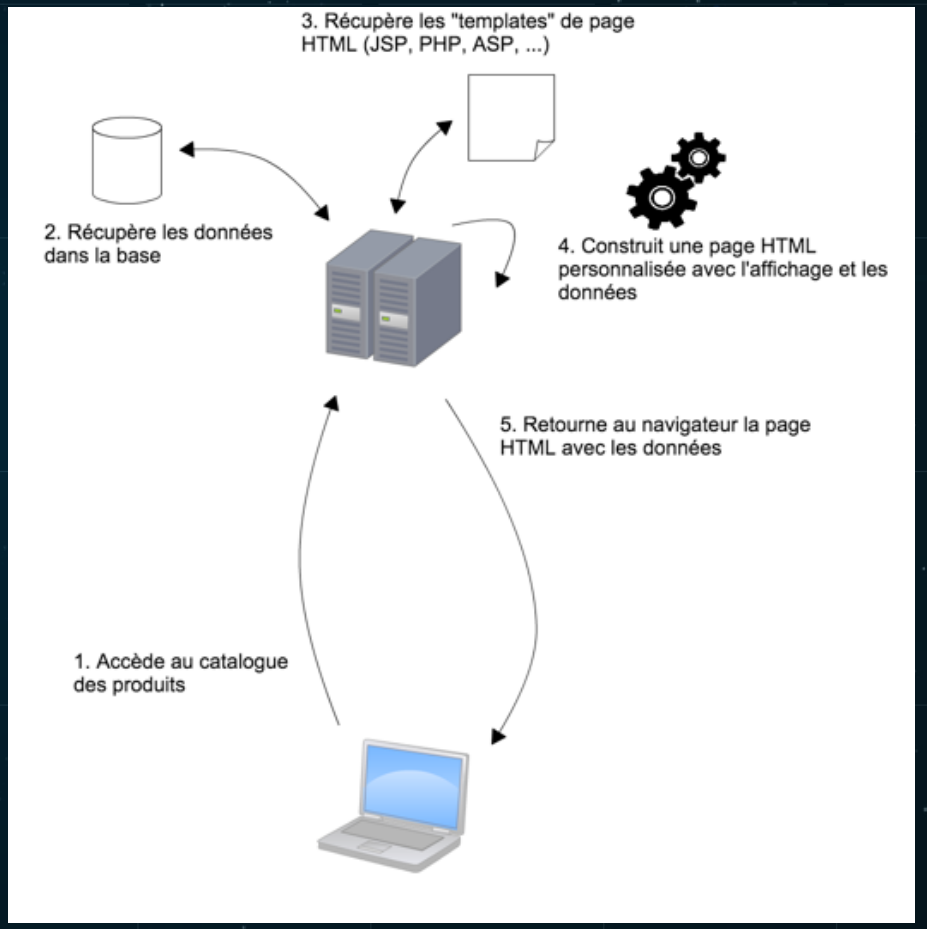
Single Page Application

Maintenant, nouveau retour en force du navigateur :

- Concept de “**SPA**”
- Frameworks: **Angular**, **VueJS**, **React**
- Permet d’apporter plus de réactivité aux interfaces et de décharger les serveurs de certaines tâches
- Le **serveur** expose des **APIs** pour accéder aux données et **l’affichage** est construit par le **navigateur**

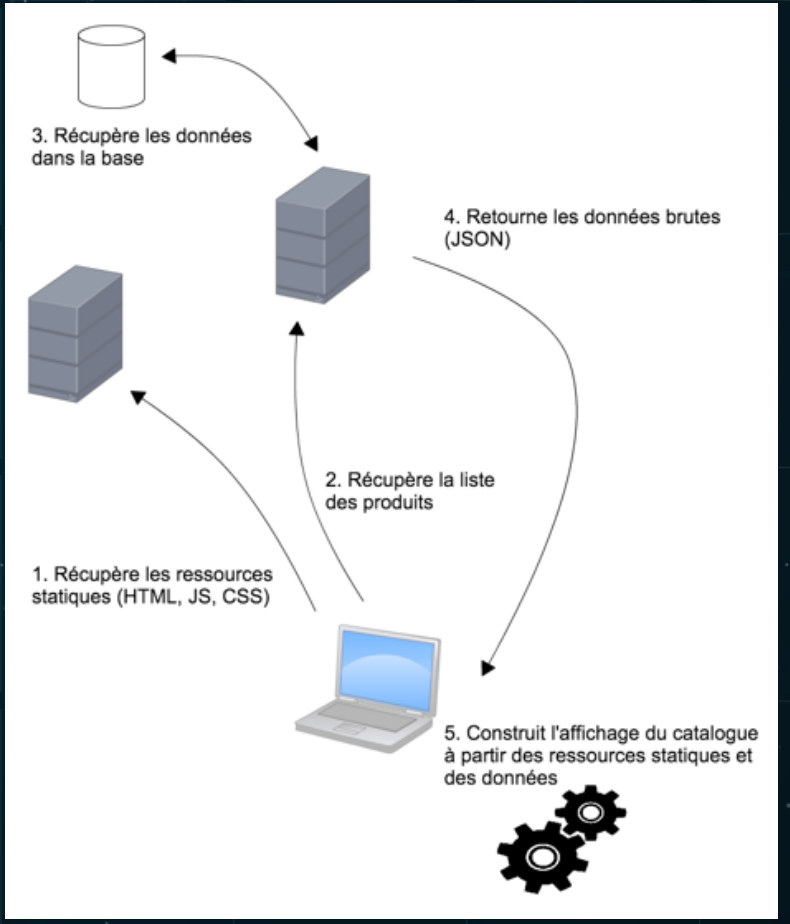
Site dynamique “classique” ...

- Les **traitements** sont effectués sur le **serveur**
- Le **navigateur** se limite à **envoyer** des **requêtes** et **afficher** les **réponses**
- ... comment faire pour intégrer un nouveau type de consommateur pour ces données (mobile, ...) ?



… vs API et SPA

- Les **traitements** sont effectués sur le **navigateur**
- Le **serveur** expose une **API** retournant les données dans un format standardisé
- **Très facile** d'intégrer de nouveaux consommateurs : mobile ou d'autres applications



Une deuxième rupture :

Les architectures distribuées

Le modèle historique

Avec l'explosion des usages de l'internet et des utilisateurs, la capacité des serveurs à gérer la charge est devenue un problème.

Comment gérer 1 000 utilisateurs simultanés ? 100 requêtes par seconde ? Répondre aux utilisateurs en moins de 1 seconde ?

On a augmenté la puissance des serveurs : mainframes, +100 Go de RAM, ...

Mais ce modèle a des limites

Google : +30 000 requêtes par secondes.

Coût des mainframes très importants.

Augmentation des puissances de CPU limitée.

Changement de paradigme vers les **architectures distribuées**.

Super-computer vs commodity hardware

Google, **Facebook**, **Twitter**, ... font tourner leurs applications sur du "**commodity hardware**" :

- Des "petits" serveurs, mais en très grand nombre.
- Cela permet de diminuer les coûts.
- De distribuer géographiquement les serveurs au plus près des utilisateurs.
- Et d'avoir une bien meilleure tolérance aux pannes.

De nouvelles contraintes

L'**architecture distribuée** implique de nouvelles contraintes :

- Des standards pour faire communiquer des applications indépendantes entre elles (**HTTP / REST**, **JSON**, ...).
- Des nouveaux langages et paradigmes pour gérer la parallélisation (**programmation fonctionnelle**).
- De nouveaux modèles d'application ("**stateless**").

L'explosion du cloud computing

Le principe du cloud computing : exploiter la puissance de calcul et de stockage de serveurs distants au travers d'Internet.

Les entreprises n'ont plus besoin de posséder les ressources informatiques, elles les louent à l'utilisation.

Par exemple : AWS, $0.015 / heure pour une instance Linux.

Permet à une entreprise de **scaler** très rapidement ses services pour s'adapter aux pics d'utilisation.

Une troisième rupture :

Les systèmes NoSQL

Les modèles relationnels

Les **bases de données relationnelles** sont les modèles les plus courants aujourd'hui.

Elles offrent de nombreux avantages :

- Interface d'accès (presque) standard : **SQL**
- Respect des contraintes **ACID** :
  - Atomicité, cohérence, isolation, durabilité.
- Des connecteurs existent pour la plupart des langages et plateformes.

Ils ont aussi leurs limites

Cependant, elles ne sont pas adaptées à toutes les situations :

- Temps de réponses dégradés sur de très gros volumes.
- Nombre d'accès concurrents limité.
- Scalabilité moins évidente.
- Le modèle relationnel n'est pas adapté à toutes les problématiques !

De nouvelles solutions apparaissent

En conséquence à ces limites des systèmes traditionnels, de grands acteurs de l'Internet ont conçu de **nouveaux systèmes**, spécifiquement adaptés à leurs besoins :

- Contraintes en terme de temps de réponse.
- Plus adaptés au déploiement distribué (**scalabilité**).
- Nouvelle représentation des données ou de la façon de les interroger (**bases orientées graphes**, **stockage orienté recherche**, ...).

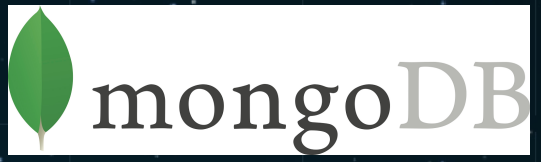
“NoSQL” : un nom unique - de nombreuses solutions

Derrière l’étiquette “**NoSQL**” existent différents outils, très différents, adaptés à des cas d’utilisations spécifiques.

A la différence des systèmes relationnels, les systèmes **NoSQL** se sont fortement spécialisés en fonction des usages.

Nous allons voir quelques exemples...

Stockage orienté document



- L'unité de stockage n'est plus une ligne composée de colonnes, mais un **document.**
- Peut être du **JSON**, de **l'XML**, ...
- **Absence de schéma** : tous les documents n'ont pas forcément la même structure.
- Avantages : **simplicité**, **scalabilité**, **performances.**
- **Pas de notion de clé étrangère** : on perd la consistance, mais on évite les jointures, coûteuses en performance.

Stockage clé-valeur

- Une sorte de “HashMap” distribuée : **on associe à une clé une valeur** qui peut être une simple chaîne de caractère ou un objet sérialisé.
- **Très performants** : conçus pour travailler principalement en mémoire.
- Le modèle de stockage / requêtage change fortement : stocker les mêmes données plusieurs fois avec des clés différentes.
- Adapté à du cache applicatif par exemple.

Stockage orienté colonne

- Les **données sont stockées par colonne** et non plus par ligne comme dans les bases relationnelles.
- Le **nombre de colonnes** d'une même table peut donc **varier d'une ligne à l'autre** : schéma plus flexible.
- Adaptés aux très gros volumes (plusieurs millions de lignes).
- Stockage orienté **insertion et accès par clé**, les mises à jour sont plus coûteuses.

Base de données orientées graphe

- Ici l'objectif n'est pas d'optimiser les performances ou la scalabilité, mais de **changer le modèle de requêtage.**
- **Stockage** des données sous forme de **noeuds** et de **liens entre ces noeuds.**
- Requêtage par parcours de **graphe.**
- Adapté pour représenter certains domaines inadaptés aux bases relationnelles : les liens d'un réseau social par exemple.

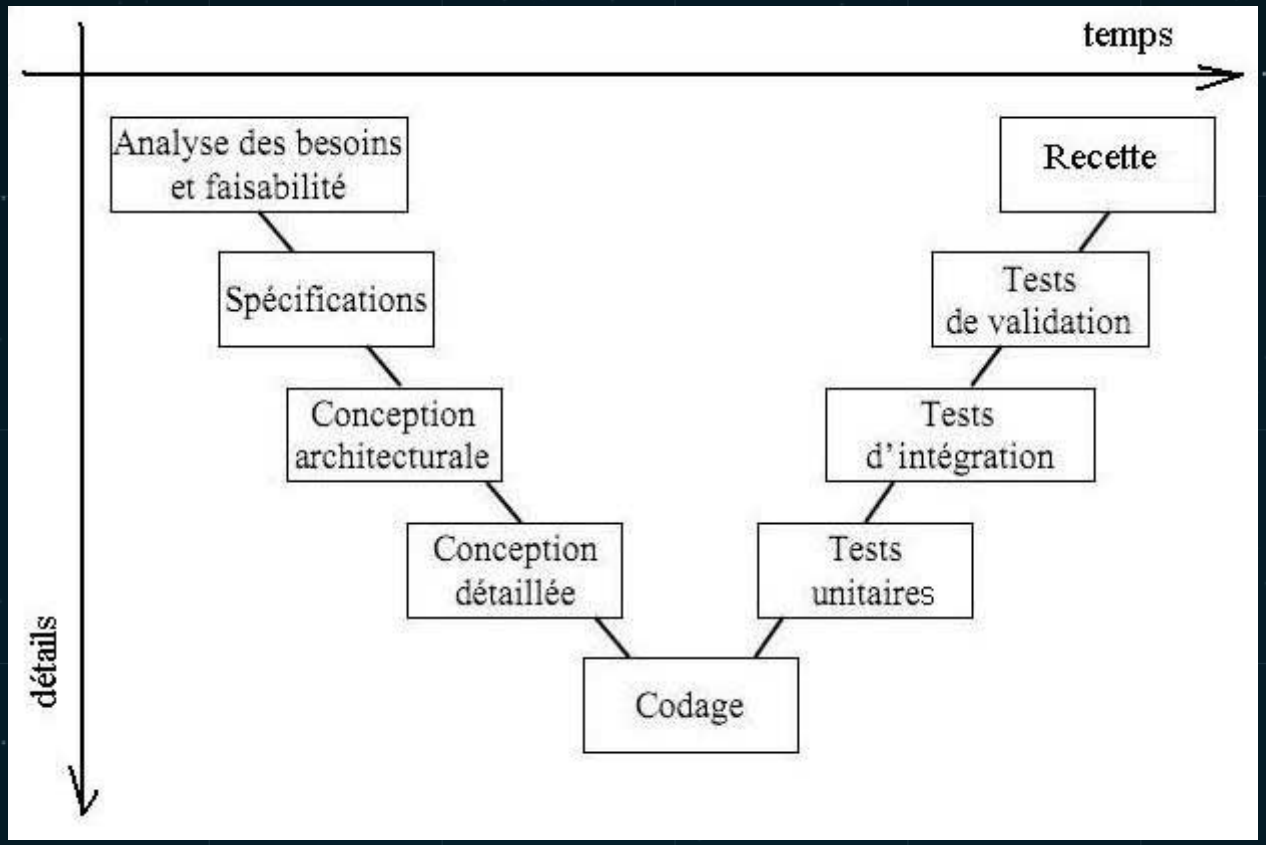
Un changement pour les applications

Ces nouveaux systèmes apportent chacun des concepts nouveaux et des contraintes différentes sur le design des applications :

- Pas de langage unifié de requêtage mais des **syntaxes spécifiques** à chaque outil.
- Nécessite de **repenser** la façon de stocker les données : dénormalisation, choisir entre optimiser les écritures ou les lectures, ...
- **Connecteurs spécifiques** à chaque système (pas de JDBC !).

Une quatrième rupture :

Les nouvelles méthodologies


temps
Analyse des besoins et faisabilité
Spécifications
Conception architecturale
Conception détaillée
Codage
Tests unitaires
Tests d'intégration
Tests de validation
Recette
détails

Cycle en V

La différence entre la théorie et la pratique...

- En **théorie**, une fois une phase terminée, elle est considérée comme définitive et ne sera plus remise en cause.
- En **pratique**... il est très fréquent de rencontrer des problèmes lors de la phase d'implémentation qui nécessite de revoir l'architecture technique ou fonctionnelle.
- De plus, **ce modèle augmente le risque** de décalage entre le besoin initial et l'application finale.
- Besoin de plus de **flexibilité** !

Les cycles courts et les méthodes agiles

La tendance est au développement avec des **cycles très courts** :

- Design **itératif**.
- Livrer **peu** à chaque fois mais livrer **souvent**.
- Obtenir du feedback plus **rapidement**.
- Permet de **minimiser** le décalage entre le besoin initial et l'application finale.

Déployer en continu

Les façons de livrer des nouvelles versions des applications évoluent en conséquence :

- Remplacer les “grosses” releases 3 fois par an par de petites releases tous les mois par exemple.
- **Moins de fonctionnalités** à chaque fois, mais **plus souvent**.
- Le **risque** est **minimisé**, la **satisfaction** des utilisateurs est **améliorée**.

Déployer en continu

Les “pure-players” vont jusqu’à déployer des nouvelles versions en production tous les jours, plusieurs fois par jours, sans que l’utilisateur ne s’en rendent compte !

Même sans aller à ces extrêmes, **livrer régulièrement** ce qui est terminé **améliore la réactivité**.

Cela nécessite des **méthodologies**, des **outils**, des **architectures adaptées**.

Quelques solutions

- **Outils d'intégration continue** pour compiler et tester automatiquement ce qui est développé.
- Les **déploiements** doivent être systématiquement **automatisés**.
- ... mais les **rollbacks** aussi ;-)
- **Docker** a fortement transformer la façon de livrer les applications.