

## Partie 5 : lien vers la présentation



<https://drive.google.com/file/d/1tJJHiae0pv04zs8FXPx1D5lqvA9TZ42Z/view?usp=sharing>

## Partie 5 - Choix des technologies

- 1. Introduction
- 2. Front-end : langages, frameworks et performances
- 3. CMS et solutions e-commerce modernes
- 4. Technologies mobiles et multiplateformes
- 5. Back-end : langages, frameworks, et architectures modernes
- 6. Outils et environnements de développement

## Partie 5 - Choix des technologies

---

- 7. Bases de données
- 8. Architectures orientées événements et intégrations
- 9. Microservices, API et conteneurisation
- 10. Tests logiciels et qualité continue
- 11. Intégration, déploiement et observabilité
- 12. Principes et patterns architecturaux

## 1 .Introduction

**Pourquoi et comment choisir une technologie ?**

- Le choix technologique influence la **scalabilité, la sécurité et la pérennité** d'une application.
- Il doit s'appuyer sur des critères objectifs :
  - Maturité et stabilité du produit
  - Taille et activité de la communauté
  - Support commercial et documentation
  - Coût total (licences, cloud, formation)
  - Compétences disponibles dans l'équipe

## 1. Introduction

- Exemple : Node.js ou Java Spring Boot ?
  - Node.js → rapidité, écosystème JavaScript
  - Spring Boot → robustesse, maturité, intégration d'entreprise
- Outils d'aide au choix :
  - [StackShare.io](#)
  - [ThoughtWorks TechRadar](#)
  - [InfoQ Trends](#)

## 2. Front-end : langages, frameworks et performances

- Html
  - Html : HyperText Markup Language, 1993, [standard W3C](#)
  - Html5 depuis 2014 :
    - support images vectorielles (SVG)
    - support audio et video
    - nouveaux éléments et attributs. Ex : canvas
    - support complet pour que JavaScript s'exécute en arrière-plan (grâce à l'API JS de HTML5).
    - MathML inline et des SVG peuvent être utilisés directement dans le texte alors que cela n'était pas possible en HTML.
    - Suppression éléments obsolètes : isindex, noframes, acronym, applet, basefont, dir, font, frame, frameset, big, center, strike, tt.
    - nouveaux types de contrôles de formulaire : les dates et heures, les e-mail, les numéros, les plages (range), les téléphones, les url ...

## 2. Front-end : langages, frameworks et performances

- Html
  - svg inline (exemple)

```
▼<svg version="1.1"  
      xmlns="http://www.w3.org/2000/svg"  
      xmlns:xlink="http://www.w3.org/1999/xlink"  
      viewBox="0,0,1024,1024">  
  ▶<desc>...</desc>  
  ▶<defs>...</defs>  
  ▶<g fill="none" fill-rule="nonzero"  
      style="mix-blend-mode:normal">...</g>  
</svg>
```

## 2. Front-end : langages, frameworks et performances

- CSS : Cascading Style Sheets, 1996, standard W3C
  - CSS3 : développement commencé en 1999  
prise en compte partielle et très lente par les navigateurs
  - quelques exemples de nouvelles fonctionnalités :
    - la négociation de style entre serveurs et agents utilisateurs (« Media Queries »)
    - le rendu web TV
    - la gestion des couleurs
  - vérifier la prise en compte : <https://caniuse.com/>
  - outils (préprocesseur) : sass : permet l'utilisation de variables

## 2. Front-end : langages, frameworks et performances

- CSS : exemple media queries

```
@media only screen and (min-width: 550px) and (max-width: 1089px) {  
    .movie-details-poster-wrapper {  
        display: flex;  
        flex-direction: column;  
        align-items: center;  
  
        .movie-details-poster {  
            margin: 20px 0;  
        }  
    }  
}
```

## 2. Front-end : langages, frameworks et performances

- Javascript : 1996, standard : ECMAScript
  - Prend en charge le contenu dynamique des pages Web
  - Langage est interprété
  - Faiblement typé
- Typescript : 2012, standard : ECMAScript
  - Langage compilé
  - Fortement typé
  - Le code TypeScript est transcompilé en JavaScript, et peut ainsi être interprété par n'importe quel navigateur web ou moteur JavaScript.

## 2. Front-end : langages, frameworks et performances

- React : bibliothèque javascript, Meta (Facebook), 2013
  - <https://fr.reactjs.org/>
  - Avantages et inconvénients :

<ul style="list-style-type: none"><li>✓ Free and open-source</li><li>✓ Easy to learn</li><li>✓ Lightweight</li><li>✓ Search engine optimization (SEO) friendly</li><li>✓ Large developer community</li><li>✓ Ideal for building user interfaces (UI)</li><li>✓ Declarative</li><li>✓ Reusable components</li><li>✓ Migration friendly</li><li>✓ Mobile browser support</li><li>✓ JavaScript Syntax Extension (JSX)</li></ul>	<ul style="list-style-type: none"><li>✗ Hard to keep up-to-date</li><li>✗ Lack of detailed documentation with the latest updates</li></ul>
--	--

## 2. Front-end : langages, frameworks et performances

- Angular : framework basé sur typescript, Google, 2016
  - <https://angular.io/>
  - Avantages et inconvénients :

<ul style="list-style-type: none"><li>✓ Free and open-source</li><li>✓ Ideal for developing Single Page Applications (SPA)</li><li>✓ Easier Document Object Model (DOM) manipulation</li><li>✓ Two-way data binding</li><li>✓ Reusable components</li><li>✓ Ideal testing framework</li><li>✓ Full-fledged solution</li><li>✓ Monolithic framework</li><li>✓ Well documented</li></ul>	<ul style="list-style-type: none"><li>✗ Better suited for more advanced code developers</li><li>✗ Performance issues</li></ul>
--	--

## 2. Front-end : langages, frameworks et performances

- Vue : framework basé sur typescript, Communauté, 2014
  - <https://vuejs.org/>
  - Avantages et inconvénients :

- ✓ Free and open-source
- ✓ Flexible and lightweight
- ✓ Easy to learn
- ✓ Fastest JS Framework available
- ✓ Two-way data binding
- ✓ Virtual Document Object Model (DOM)
- ✓ Good for developing Single Page Applications (SPA)

- ✗ Fewer job opportunities than its competitors
- ✗ Lack of constant updates and bug fixes due to a minor developer team
- ✗ Not ideal for large-scale projects

## 2. Front-end : langages, frameworks et performances

---

- Next.js (React) : rendu hybride (SSG + SSR), API Routes, Server Components.
- Nuxt.js (Vue) : équivalent Vue, compatible avec CMS headless.
- SvelteKit : compilation native, ultra-léger.
- Astro : génération statique, contenu partiel chargé dynamiquement.
  
- Angular vs React : <https://www.youtube.com/watch?v=qNuttNYctjM>

## 2. Front-end : langages, frameworks et performances

- SPA : Single Page Application : implémentation d'application web qui ne charge qu'un seul document web, puis met à jour le contenu du corps de ce document via des API JavaScript telles que XMLHttpRequest et Fetch lorsqu'un contenu différent doit être affiché.
- Responsive : “Responsive design”. Cela consiste à rendre un site web accessible et adaptable à tous les devices : tablettes, smartphones, ordinateurs, tv.
- PWA : Progressive Web App : Ce type d'applications tente de combiner les fonctionnalités offertes par la plupart des navigateurs modernes avec les avantages de l'expérience offerte par les appareils mobiles

## 2. Front-end : langages, frameworks et performances

### Focus sur les Rendering Patterns

- Les Rendering Patterns définissent où et quand le HTML est généré : côté client, serveur, ou build.
- Objectif : trouver le bon équilibre entre performance, SEO, et coût serveur.
- Principaux patterns :
  - Client-Side Rendering (CSR)
  - Server-Side Rendering (SSR)
  - Static Rendering (SSG)
  - Incremental Static Generation (ISG)
  - Streaming SSR
  - Progressive Hydration
  - Selective Hydration
- Un framework moderne combine souvent plusieurs patterns selon les pages ou les routes.

## 2. Front-end : langages, frameworks et performances

Focus sur les Rendering Patterns : Client-Side Rendering (CSR)

- Principe :
  - Le navigateur reçoit une page vide + un bundle JavaScript.
  - Le rendu (DOM, interactions, données) est généré entièrement côté client.
- Exemples : applications SPA classiques (React, Vue, Angular).
- Avantages :
  - Très interactif.
  - Peu de charge serveur.
  - Architecture simple à déployer (CDN).
- Limites :
  - Mauvais SEO initial.
  - Temps d'affichage initial plus long (First Paint).
- Approprié pour des applications internes ou à usage authentifié.

## 2. Front-end : langages, frameworks et performances

Focus sur les Rendering Patterns : Server-Side Rendering (SSR) et Static Rendering (SSG)

- Server-Side Rendering (SSR)
  - Le HTML complet est généré côté serveur à chaque requête.
  - Le navigateur reçoit du contenu déjà rendu.
  - Le JavaScript “hydrate” ensuite la page côté client.
  - exemple : <https://www.patterns.dev/react/server-side-rendering/>
- Static Rendering (SSG)
  - Le HTML est pré-généré au moment du build, puis servi tel quel depuis un CDN.
  - exemple : <https://www.patterns.dev/react/static-rendering/>
- Exemples : blogs, documentations, sites à contenu fixe.
- SSR = contenu toujours frais / SSG = performance maximale

## 2. Front-end : langages, frameworks et performances

Focus sur les Rendering Patterns : Incremental Static Generation (ISG)

- Principe :
  - Variante du SSG : certaines pages sont générées à la demande, puis mises en cache.
  - Les pages non encore générées sont construites “on the fly” et servies ensuite comme statiques.
- Avantages :
  - Performance du statique + actualisation incrémentale.
  - Évite le rebuild tout le site à chaque changement.
- Exemples :
  - Next.js : getStaticProps + revalidate
  - Nuxt 3 “on-demand revalidation”
- Idéal pour les sites à large catalogue (e-commerce, actualités).

## 2. Front-end : langages, frameworks et performances

### Focus sur les Rendering Patterns : Progressive Hydration

- Principe :
  - Après le rendu initial (SSR ou SSG), le JavaScript est chargé progressivement pour chaque composant.
  - Les parties visibles de la page sont hydratées d'abord, les autres ensuite.
- Avantages :
  - Réduit le temps d'interactivité (TTI).
  - Optimise le rendu pour les pages longues.
  -
- Exemples :
  - React 18 avec ReactDOM.createRoot() (concurrent features).
  - Vue 3 avec hydration progressive expérimentale.
- Approche utilisée dans Next.js, Remix et Astro pour le rendu “islands architecture”.
- exemple : <https://www.patterns.dev/react/progressive-hydration/>

## 2. Front-end : langages, frameworks et performances

Focus sur les Rendering Patterns : Selective Hydration et Streaming SSR

- Selective Hydration
  - Le framework choisit quels composants hydrater selon leur priorité (visibilité, interaction).
  - Permet d'éviter d'hydrater toute la page.
  - exemple : <https://www.patterns.dev/react/react-selective-hydration/>
- Streaming SSR
  - Le serveur envoie le HTML par flux dès qu'il est prêt (sans attendre tout le rendu).
  - Améliore le Time To First Byte (TTFB) et la perception de vitesse.
  - Le Streaming SSR est la base des architectures isomorphiques modernes.  
  
architectures isomorphiques : architecture où le même code peut être exécuté à la fois sur le client et le serveur.
  - exemple : <https://www.patterns.dev/react/streaming-ssr/>

## 2. Front-end : langages, frameworks et performances

### Focus sur les Rendering Patterns : cas d'utilisation

Cas d'usage	Pattern recommandé	Exemple concret
Application SaaS / SPA interne	CSR	Tableau de bord React
Blog / Documentation	SSG	Astro, Next.js static export
E-commerce / actualités	ISG	Next.js avec revalidation
Application universelle SEO + réactivité	SSR + Progressive Hydration	Next.js 14, Nuxt 3
Plateforme haute performance	Streaming SSR	React 18 + Server Components

## 2. Front-end : langages, frameworks et performances

### Focus sur les Rendering Patterns : support par les frameworks

Framework	CSR	SSR	SSG	ISG	Streaming SSR	Progressive Hydration	Selective Hydration
React	✓	(via Next.js)	(via Next.js/Gatsby)	(via Next.js)	✳️ (React 18+)	✓	✓
Angular	✓	✓ (Angular Universal)	✳️ limité	✗	✳️ partiel	✳️ expérimental	✗
Vue.js	✓	(via Nuxt)	(via Nuxt)	(via Nuxt 3)	✳️ (Nuxt 3)	✳️ expérimental	✗
Next.js	✓	✓	✓	✓	✓	✓	✓
Nuxt 3	✓	✓	✓	✓	✓	✳️ expérimental	✗

💡 Légende :

✓ = support natif   ✳️ = partiel / expérimental   ✗ = non supporté

💡 Next.js et Nuxt 3 sont aujourd'hui les frameworks les plus complets sur le plan des stratégies de rendu.

## 2. Front-end : langages, frameworks et performances

### Accessibilité et performance web

- Accessibilité (A11y) :
  - Respect des standards WAI-ARIA (Web Accessibility Initiative - Accessible Rich Internet Applications)
  - [WAI-ARIA basics](#)

## 2. Front-end : langages, frameworks et performances

### Accessibilité et performance web

- Performance : Core Web Vitals (Google) :
  - LCP (Largest Contentful Paint) : temps de chargement principal
  - INP (Interaction to Next Paint) : interactivité
  - CLS (Cumulative Layout Shift) : stabilité visuelle

détails :  
<https://developers.google.com/search/docs/appearance/core-web-vitals?hl=fr>
- Outils :
  - [Lighthouse](#)
  - [WebPageTest](#)
  - Chrome DevTools Performance Tab
- Bonnes pratiques : lazy loading, compression, cache HTTP, images [WebP](#).

## 3. CMS et e-commerce modernes

- Définition : “Content Management System” ou “Système de gestion de contenus” en français.  
Un CMS est donc une application ou un logiciel qui utilise une base de données pour gérer tout le contenu d'un site.
- Quelques CMS :
  - Wordpress
  - Joomla
  - Drupal
  - Typo3
  - Jahia

## 3. CMS et e-commerce modernes

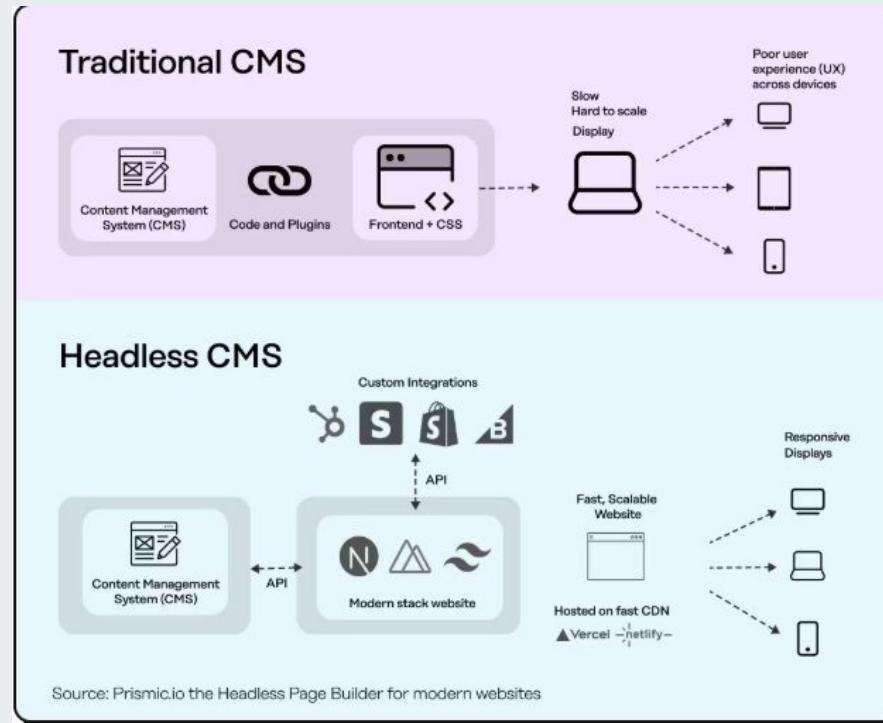
- Fonctionnalités d'une solution e-commerce :
  - catalogue produits
  - paiement en ligne
  - facturation
  - expédition
- Quelques solutions e-commerce :
  - Prestashop
  - Shopify
  - Magento
  - WooCommerce

## 3. CMS et e-commerce modernes

### Les CMS Headless

- Principe : le contenu est géré via API, le front est libre (React, Vue, mobile).
- Exemples : Strapi, Sanity, Contentful, Directus.
- Avantages :
  - Indépendance du front-end
  - Déploiement multi-plateformes
  - Scalabilité et API-first
- Cas d'usage : sites vitrines, PWA, contenus omnicanaux.

## 3. CMS et e-commerce modernes



## 4 . Technologies mobiles et multi-plateformes

- React Native : développement d'applications pour Android, iOS et UWP (Universal Windows Platform).  
Permet aux développeurs d'utiliser React avec les fonctionnalités natives de ces plateformes.
- Flutter : Google, 2017
  - avec le langage Dart
  - Android, iOS, Linux, Mac, Windows, Google Fuchsia et le web
  - compilé en natif
- Kotlin Multiplatform (JetBrains) : mutualisation logique métier Android/iOS.
- Capacitor / Ionic / Expo : wrappers hybrides pour web et mobile.

## 5. Back-end : langages, frameworks, et architectures modernes

Langage	Framework / bibliothèque	Exemples connus
JavaScript (1996)	Node.js	Netflix Facebook
Python (1991)	Django Flask	Instagram Google Spotify
Ruby (1995)	Ruby on Rails	GitHub Zendesk Basecamp

L'ordre de présentation ne correspond pas à l'ordre de popularité des langages.

## 5. Back-end : langages, frameworks, et architectures modernes

Langage	Framework / bibliothèque	Exemples connus
PHP (1994)	Laravel Cake PHP Symfony	Yahoo Wordpress Wikipedia
Java (1996)	Spring, Hibernate	Hadoop Jenkins
C# (2001)	.NET Unity	Visual Studio Windows Installer
Perl (1987)	Catalyst	IMDB Amazon BBC

## 5. Back-end : langages, frameworks, et architectures modernes

Langage	Framework / bibliothèque	Exemples connus
C++ (1985)	APR (Apache Portable Runtime) ASL (Adobe)	Adobe Photoshop Office Mozilla Firefox
Kotlin (2011)	Javalin KTor Spark	Trello Evernote
Scala (2004)	Play Akka Chaos	Lichess Apache Kafka

Exemples : <https://www.geeksforgeeks.org/hello-world-in-30-different-languages/>

## 5. Back-end : langages, frameworks, et architectures modernes : Index Tiobe (1 / 2)

Oct 2025	Oct 2024	Change	Programming Language	Ratings	Change
1	1		 Python	24.45%	+2.55%
2	4		 C	9.29%	+0.91%
3	2		 C++	8.84%	-2.77%
4	3		 Java	8.35%	-2.15%
5	5		 C#	6.94%	+1.32%
6	6		 JavaScript	3.41%	-0.13%
7	7		 Visual Basic	3.22%	+0.87%
8	8		 Go	1.92%	-0.10%
9	10		 Delphi/Object Pascal	1.86%	+0.19%
10	11		 SQL	1.77%	+0.13%

source : <https://www.tiobe.com/tiobe-index/>

## 5. Back-end : langages, frameworks, et architectures modernes : Index Tiobe (2 / 2)

- Règles de calcul du classement :

“The TIOBE Programming Community index is an indicator of the popularity of programming languages. The index is updated once a month. The ratings are based on the number of skilled engineers world-wide, courses and third party vendors. Popular search engines such as Google, Bing, Yahoo!, Wikipedia, Amazon, YouTube and Baidu are used to calculate the ratings. It is important to note that the TIOBE index is not about the best programming language or the language in which most lines of code have been written.”

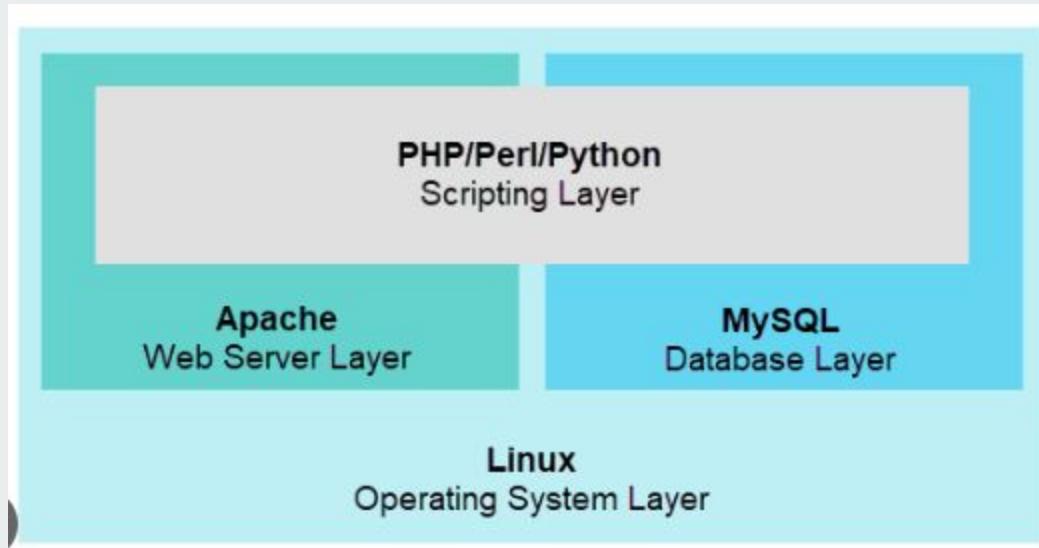
- Certains langages qui ne figurent pas dans le classement Tiobe restent très utilisés en entreprise. Exemple : Cobol  
<https://www.lemagit.fr/conseil/Strategies-pour-garder-Cobol-et-moderniser-ses-mainframes>

## 5. Back-end : langages, frameworks, et architectures modernes : les technologies full-stack

- LAMP :
  - Linux, Apache, MySql, Php
  - Développement Web
  - alternatives à PHP : Perl, Python
  - alternative à Apache : Nginx
  - alternative à MySql : MariaDb
  - en 2021, représente 30% des sites web en service

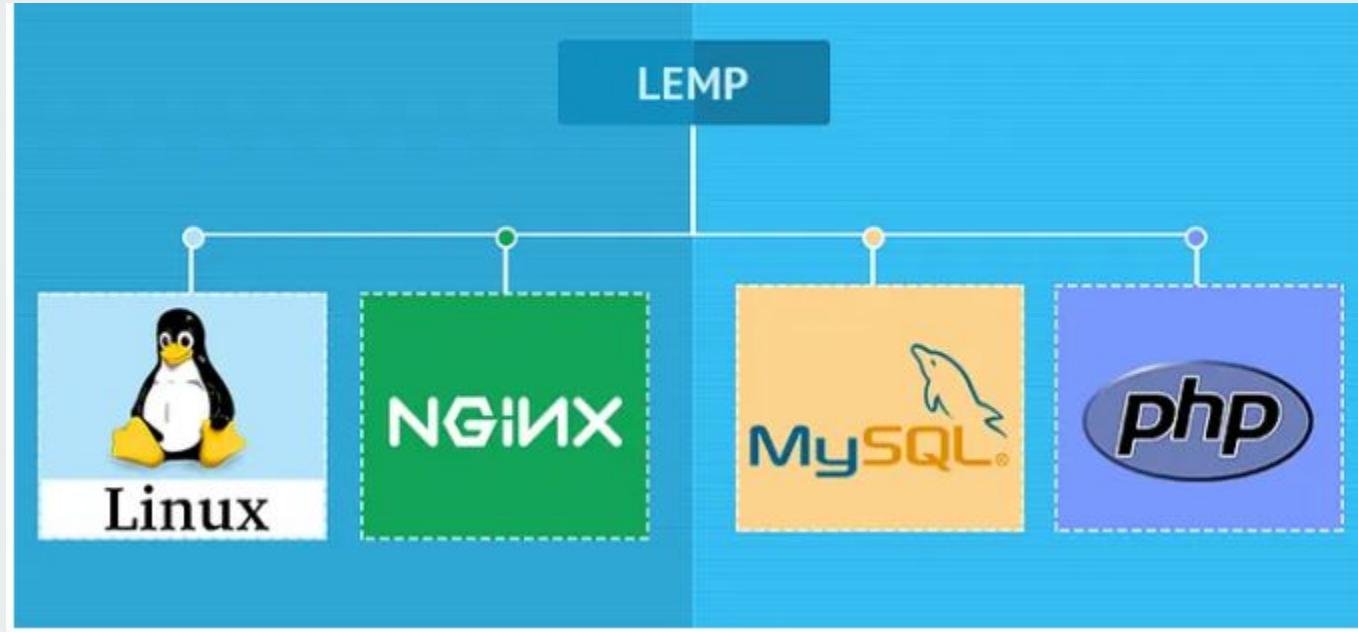
## 5. Back-end : langages, frameworks, et architectures modernes : les technologies full-stack

- LAMP



## 5. Back-end : langages, frameworks, et architectures modernes : les technologies full-stack

- LEMP (E pour Engine-X)

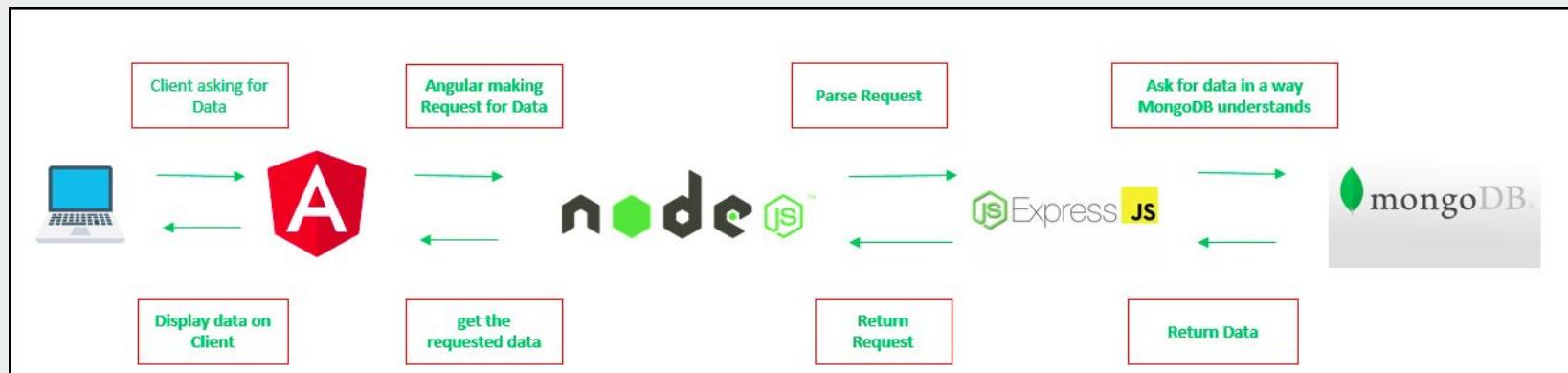


## 5. Back-end : langages, frameworks, et architectures modernes : les technologies full-stack

- MEAN :
  - MongoDB, Express.js, Angular, Node.js
  - Développement Web
  - MongoDB : système de gestion de base de données orienté documents
  - Node.js : plateforme logicielle libre en JavaScript, permettant de développer des applications back-end avec de très fortes capacités de montée en charge
  - Express.js : framework le plus utilisé pour le développement sur Node.js

## 5. Back-end : langages, frameworks, et architectures modernes : les technologies full-stack

- MEAN :



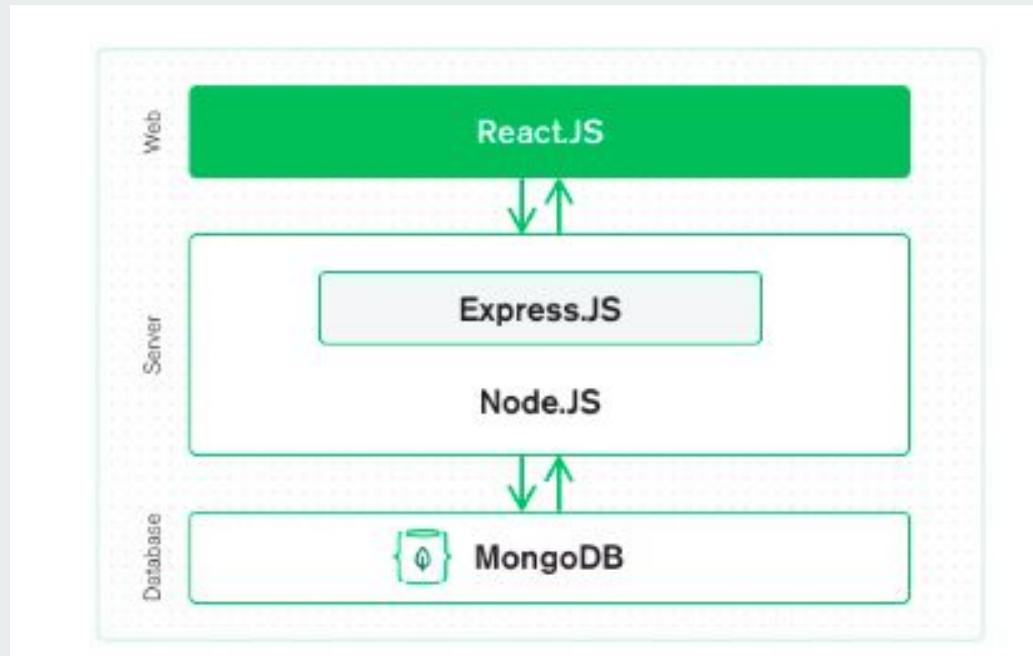
## 5. Back-end : langages, frameworks, et architectures modernes : les technologies full-stack

---

- MERN
  - MongoDB, Express.js, React, Node.js
  - Développement Web

## 5. Back-end : langages, frameworks, et architectures modernes : les technologies full-stack

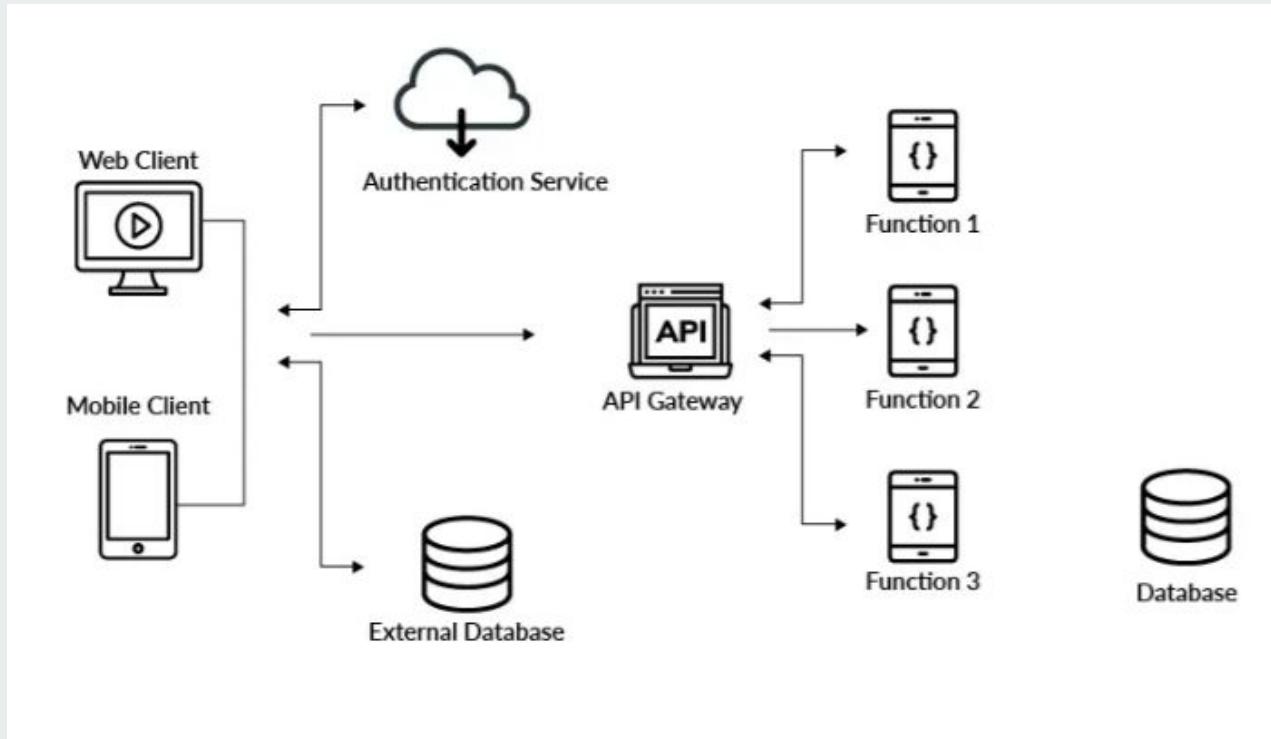
- MERN



## 5. Back-end : langages, frameworks, et architectures modernes : Serverless et BaaS

- Serverless : exécution à la demande sans serveur géré.
  - AWS Lambda, Azure Functions, Google Cloud Functions , Vercel
  - Facturation à l'usage, scalabilité automatique.
- BaaS (Backend as a Service) : backend complet en quelques clics.
  - Firebase, Supabase, Appwrite.
  - Authentification, stockage, temps réel.
- Inconvénients : vendor lock-in, cold start, monitoring plus complexe.

## 5. Back-end : langages, frameworks, et architectures modernes : focus Serverless



## 5. Back-end : langages, frameworks, et architectures modernes : focus AWS Lambda

- AWS Lambda : principes et fonctionnement
- Principes :
  - AWS Lambda permet d'exécuter du code sans serveur dédié.
  - « Vous fournissez le code, AWS gère l'infrastructure. »
- Le code est exécuté à la demande, en réponse à un événement :
  - HTTP via API Gateway
  - Upload S3
  - Message SQS / SNS
  - Cron (EventBridge / CloudWatch)
- Fonction = unitaire, courte, isolée.
- Runtime supportés : Java, Go, PowerShell, Node.js, C#, Python, and Ruby

## 5. Back-end : langages, frameworks, et architectures modernes : focus AWS Lambda

- Avantages :
  - Aucune gestion de serveur / scaling auto.
  - Facturation à l'exécution (ms).
  - Sécurité et isolation natives AWS.
- Limites :
  - Temps de cold start (~100 ms à plusieurs s).
  - Exécution limitée (timeout 15 min).
  - Pas de stockage local permanent.

## 5. Back-end : langages, frameworks, et architectures modernes : focus AWS Lambda

- AWS Lambda : cas d'usage - autres exemples

Cas d'usage	Description	Exemple
<b>API REST légère</b>	Appels HTTP sans backend permanent	API produits / utilisateurs
<b>Traitement data</b>	Fonction déclenchée par S3 ou Kinesis	Redimensionnement d'images, parsing CSV
<b>Automatisation / ETL</b>	Lambda + EventBridge = cron sans serveur	Nettoyage BDD chaque jour
<b>Chatbot / Webhook</b>	Intégration avec Slack, Telegram, etc.	Réponses instantanées
<b>Back-office événementiel</b>	Reaction à des événements métiers	Maj stock, génération PDF

## 6. Outils et environnements de développement : versionning

- Définition : Le contrôle de version, également appelé contrôle de source, désigne la pratique consistant à suivre et à gérer les changements apportés au code d'un logiciel.
- Autres termes utilisés : VCS, SCM
- Outils les plus connus : Git, Subversion (Svn)
- Git :
  - logiciel de gestion de versions décentralisé.
  - libre et gratuit, créé en 2005 par Linus Torvalds
- Repo interne ou Cloud : Github, Gitlab, Bitbucket

## 6. Outils et environnements de développement : prototypage

- Invision
- Figma
- Sketch
- Balsamiq

## 6. Outils et environnements de développement : développement

- Editeurs de texte avancés : notepad++, ultraedit
- IDE Editeurs et open-source :
  - Jetbrains : IntelliJ, WebStorm, PhpStorm, PyCharm, [Fleet](#)
  - Visual Studio Code
  - Eclipse Foundation : Eclipse IDE
  - [Spring Tools](#)
  - Apache Netbeans
  - [Github Codespaces](#)
  - PC Soft : WinDev, WebDev

## 6. Outils et environnements de développement : IA

- Outils de développement basé sur l'IA :

- [GitHub Copilot](#)
- AWS CodeWhisperer
- Microsoft IntelliCode
- Tabnine
- Claude Code
- Cursor AI

Principes :

- multi-langages
- intégré dans les IDE courants
- autocompletion ou proposition de code à partir d'un commentaire

## 7. Bases de données

- 2 grandes familles de bases de données ou système de gestion de bases de données :
  - SGBDR : bases de données relationnelles
  - SGBD NoSql

## 7. Bases de données : SGBDR

- Caractéristiques
  - la prise en charge de l'atomicité, de la cohérence, de l'isolation et de la durabilité des transactions (propriétés regroupées sous l'acronyme ACID : Atomicity, Consistency, Isolation, Durability)  
Ces propriétés garantissent l'exécution correcte de toutes les transactions ou, en cas d'échec de celles-ci, le retour d'une base de données à son état antérieur.
- Contraintes et limitations
  - la structure doit être définie avant utilisation
  - la scalabilité horizontale est difficile

## 7. Bases de données : SGBDR

---

- MySQL : open-source, 1995
- MariaDB : open-source, 2009
- Oracle : éditeur, 1980
- PostgreSQL : open-source, 1989
- Microsoft Sql Server : éditeur, 1989
- IBM Db2 : éditeur, 1983

## 7. Bases de données : NoSql

- 4 catégories :
  - Clé-valeur
  - Document
  - Colonnes
  - Graph
- Caractéristiques
  - stockage de contenu non structuré
  - pas de structure pré-définie à créer
  - scalabilité horizontale plus facile à obtenir
  - très hautes performances de lecture

## 7. Bases de données : NoSql

---

- MongoDB : document, open-source, 2009
- Redis : clé - valeur, open-source, 2009
- Elasticsearch : document, open-source, 2010
- Cassandra : colonne, open-source, 2008
- Neo4j : graph, open-source, 2007
- CouchBase : document, open-source, 2011
- MemCached : clé - valeur, open-source, 2003

## 7. Bases de données : classement

Rank Nov 2025	DBMS	Database Model	Score		
			Nov 2025	Oct 2025	Nov 2024
1.	1. Oracle	Relational, Multi-model 	1239.78	+27.01	-77.23
2.	2. MySQL	Relational, Multi-model 	865.82	-13.84	-151.98
3.	3. Microsoft SQL Server	Relational, Multi-model 	718.87	+3.82	-80.94
4.	4. PostgreSQL	Relational, Multi-model 	651.36	+8.16	-2.97
5.	5. MongoDB 	Multi-model 	371.68	+3.66	-29.25
6.	6.  7. Snowflake	Relational	197.84	-0.81	+55.35
7.	7.  6. Redis	Key-value, Multi-model 	145.09	+2.76	-3.55
8.	8.  14. Databricks	Multi-model 	131.50	+2.70	+45.04
9.	9. IBM Db2	Relational, Multi-model 	119.28	-3.10	-2.47
10.	10.  8. Elasticsearch	Multi-model 	113.97	-2.69	-17.67
11.	 12.  10. SQLite	Relational	104.19	-0.36	+4.71
12.	 11.  11. Apache Cassandra	Wide column, Multi-model 	102.71	-2.45	+5.00
13.	13.  15. MariaDB 	Relational, Multi-model 	87.36	-0.41	+4.66
14.	14.  12. Microsoft Access	Relational	78.29	-2.50	-13.03
15.	 16.  16. Microsoft Azure SQL Database	Relational, Multi-model 	76.38	+0.90	-0.15
16.	 18.  13. Splunk	Search engine	76.10	+2.22	-12.37
17.	 15. 17. Amazon DynamoDB	Multi-model 	75.78	-0.13	+3.38
18.	 17. 18. Apache Hive	Relational	72.87	-2.35	+21.37
19.	19. 19. Google BigQuery	Relational	62.79	-0.42	+12.20
20.	20.  21. Neo4j	Graph	52.36	-0.15	+9.66

- “The DB-Engines Ranking ranks database management systems according to their popularity. The ranking is updated monthly.”
- source :  
<https://db-engines.com/en/ranking>

## 7. Bases de données : partitioning, sharding

- 2 types de partitioning : vertical et horizontal

### Vertical Partition

Original Table			
CustomerID	Name	Email	Total Purchases
1	ABC	abc@gmail.com	2000
2	PQR	pqr@gmail.com	3000
3	XYZ	xyz@gmail.com	4000
4	LMN	lmn@gmail.com	5000
5	JKL	jkl@gmail.com	6000

Partition 1

CustomerID	Name	Email
1	ABC	abc@gmail.com
2	PQR	pqr@gmail.com
3	XYZ	xyz@gmail.com
4	LMN	lmn@gmail.com
5	JKL	jkl@gmail.com

Partition 2

CustomerID	Total Purchases
1	2000
2	3000
3	4000
4	5000
5	6000

### Horizontal Partition / Database Sharding

Original Table			
CustomerID	Name	Email	Total Purchases
1	ABC	abc@gmail.com	2000
2	PQR	pqr@gmail.com	3000
3	XYZ	xyz@gmail.com	4000
4	LMN	lmn@gmail.com	5000
5	JKL	jkl@gmail.com	6000

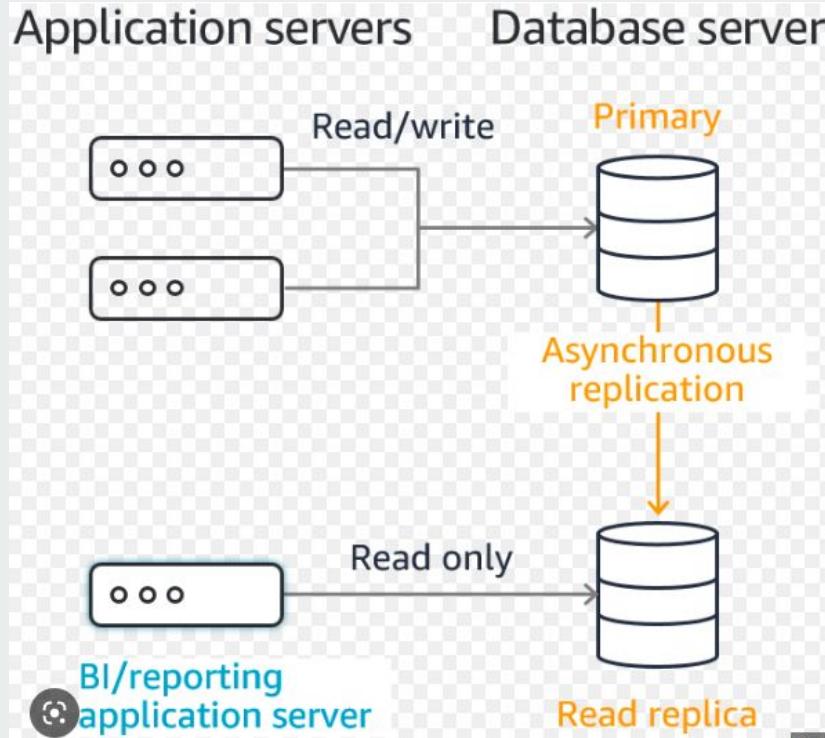
Shard 1

CustomerID	Name	Email	Total Purchases
1	ABC	abc@gmail.com	2000
2	PQR	pqr@gmail.com	3000
3	XYZ	xyz@gmail.com	4000

Shard 2

CustomerID	Name	Email	Total Purchases
4	LMN	lmn@gmail.com	5000
5	JKL	jkl@gmail.com	6000

## 7. Bases de données : réPLICATION



- cas d'utilisation d'une réPLICATION en read-only : décharger la base principale d'une partie des connexions en lecture seule

## 7. Bases de données : sql / nosql

- Les SGBDR tendent à évoluer en intégrant des fonctionnalités “NoSql”.  
ex : stockage en mémoire, stockage contenu non structuré
- En fonction du modèle d’architecture retenu, le choix de la base de données ne sera pas unique mais plusieurs bases de données seront choisies en fonction des cas d’usage

## 7. Bases de données : sql / nosql

- NewSQL : scalabilité horizontale + transactions ACID.
  - CockroachDB, YugabyteDB.
- Time-Series DB : optimisées pour données temporelles (IoT, logs).
  - InfluxDB, TimescaleDB.
- Federated Data / Data Mesh :
  - Requêtes multi-sources (API, SGBDR, NoSQL).
- Cloud DB : AWS Aurora, Google Firestore, [PlanetScale](#).

## 7. Bases de données : sql / nosql

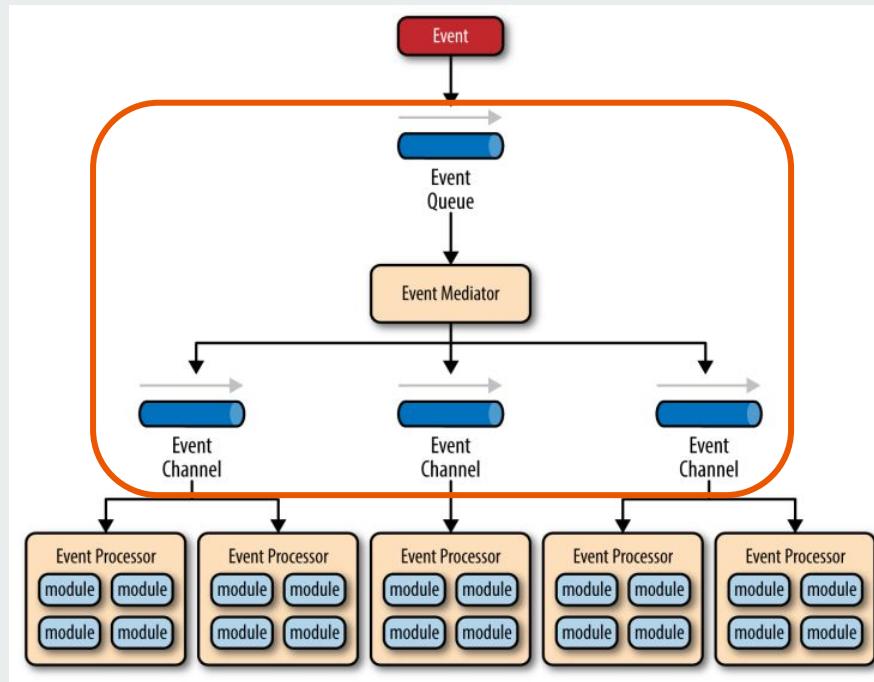
- NewSQL : scalabilité horizontale + transactions ACID.
  - CockroachDB, YugabyteDB.
- Time-Series DB : optimisées pour données temporelles (IoT, logs).
  - InfluxDB, TimescaleDB.
- Federated Data / Data Mesh :
  - Requêtes multi-sources (API, SGBDR, NoSQL).
  - GraphQL
- Cloud DB : AWS Aurora, Google Firestore, [PlanetScale](#).

## 7. Bases de données : vidéos

- PostgreSQL vs Mysql : <https://www.youtube.com/watch?v=1Wj0fzTJb6c>
- AWS RDS Database Engines: MySQL, PostgreSQL, Oracle & SQL Server:  
<https://www.youtube.com/watch?v=xvbCTmbxfwA>
- Bdd pour app Mobile : <https://www.youtube.com/watch?v=idIG-jFbDA>

## 8. Architecture orientées événements et intégrations

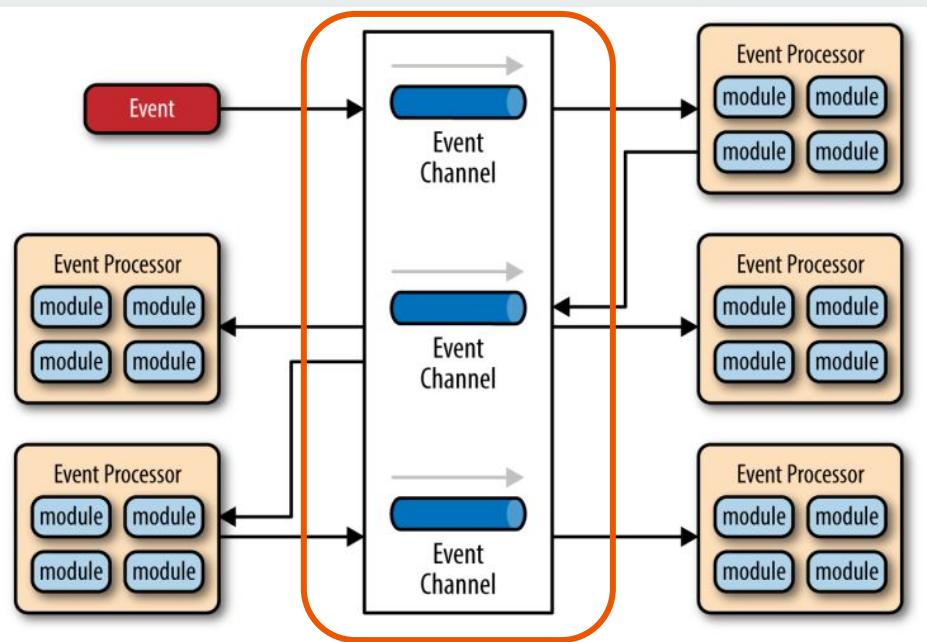
## Les technologies impliquées dans le modèle d'architecture orientée événements



Variante “Médiateur”

- Event Queue
- Event Mediator
- Event Channel

## Les technologies impliquées dans le modèle d'architecture orientée événements



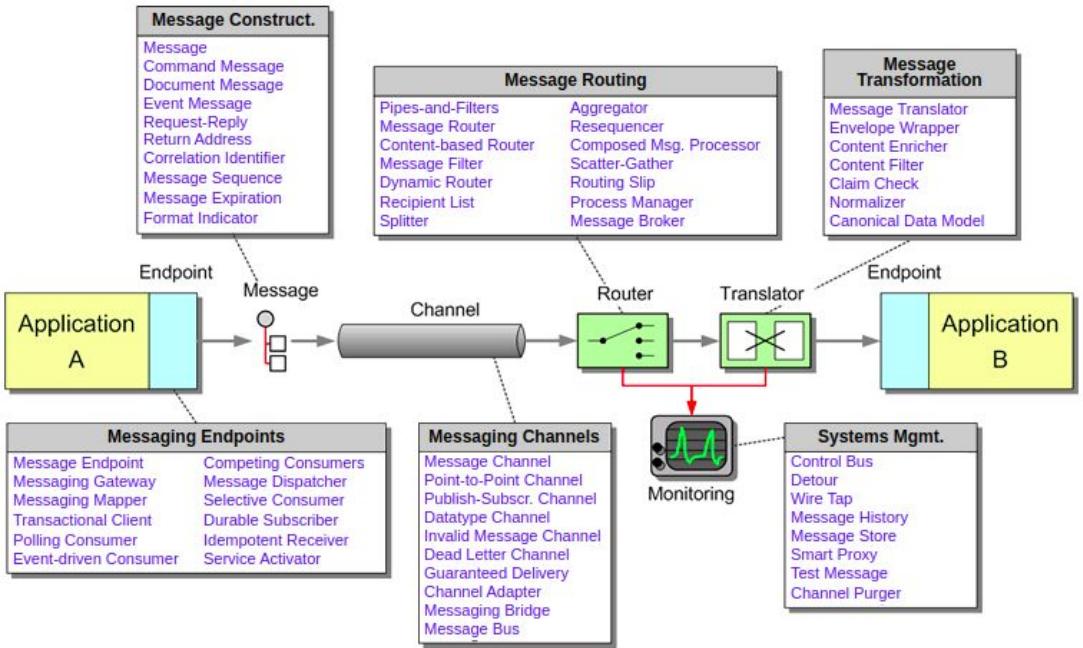
Variante “Courtier”

- Event Channel
- Event Broker

## Event Mediator, Event queue, Event channel

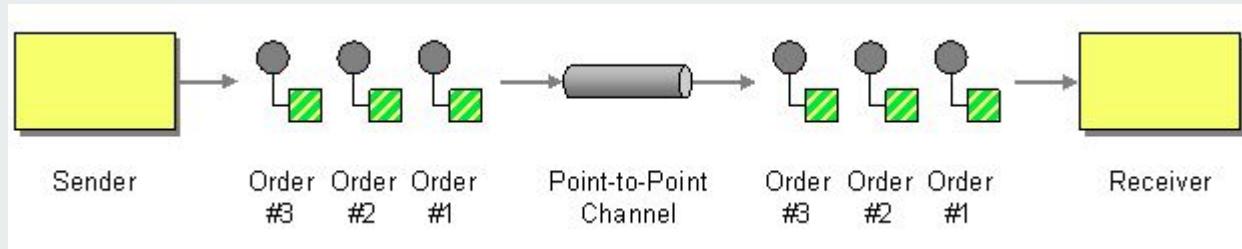
- Event mediator : implémentation EIP
  - Apache Camel
  - Spring Integration
  - Mule ESB
- Event queue, Event channel, Event broker
  - ActiveMq
  - RabbitMq
  - IBM MQ
  - Aws SQS, AWS SNS

## Focus sur EIP



- EIP : Entreprise Integration Patterns  
<https://www.enterpriseintegrationpatterns.com/>
- 65 “messaging patterns”
- livre publié en 2003, toujours d’actualité

## EIP : Point-to-Point Channel

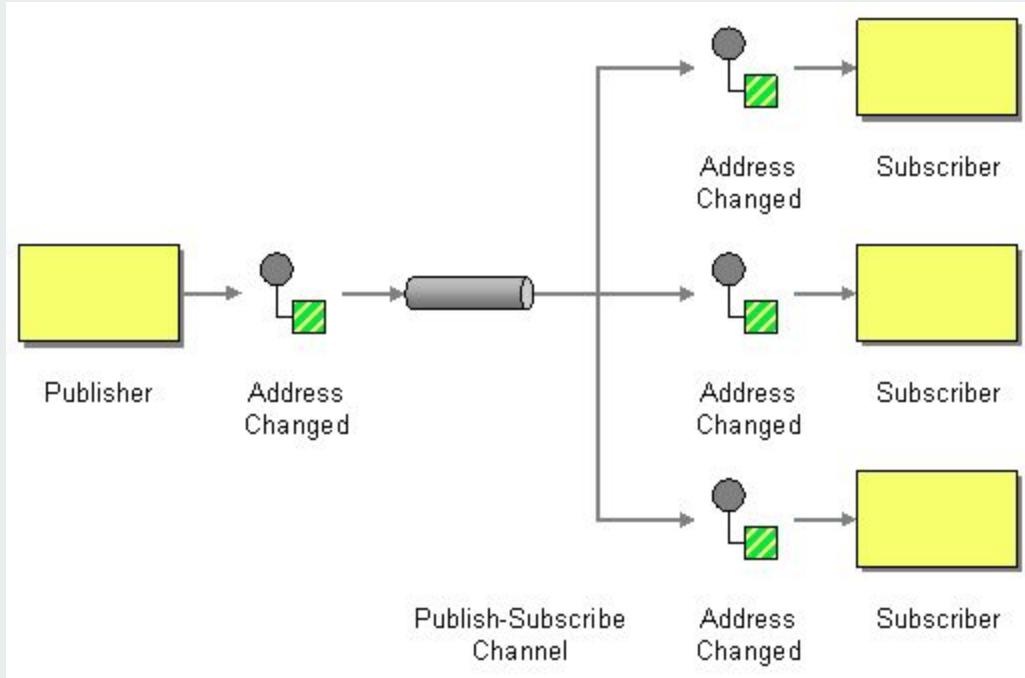


Un canal point à point garantit qu'un seul récepteur consomme un message donné.

Si le canal a plusieurs récepteurs, un seul d'entre eux peut consommer avec succès un message particulier. Si plusieurs récepteurs tentent de consommer un même message, le canal garantit qu'un seul d'entre eux y parvient, de sorte que les récepteurs n'ont pas à se coordonner entre eux.

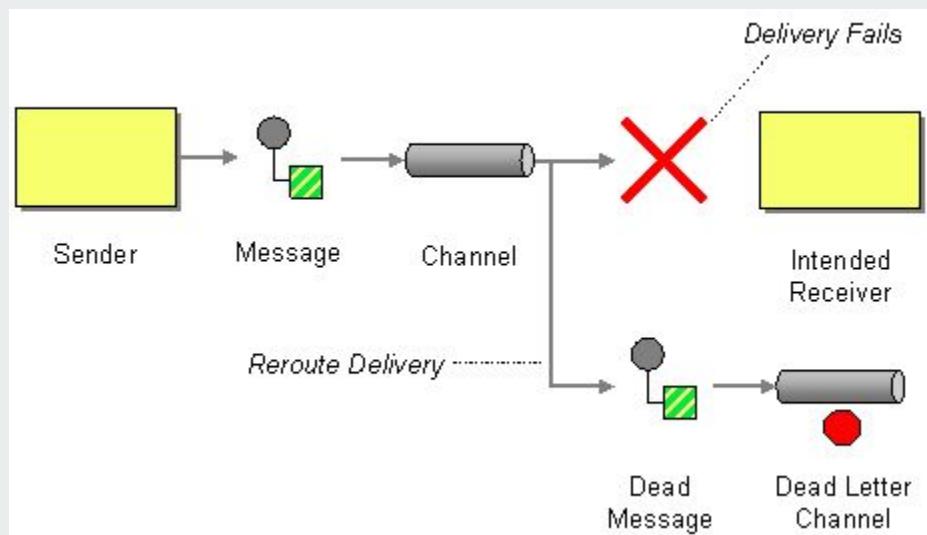
Le canal peut toujours avoir plusieurs récepteurs pour consommer plusieurs messages simultanément, mais un seul récepteur peut consommer un message.

## EIP : Publish-Subscribe Channel



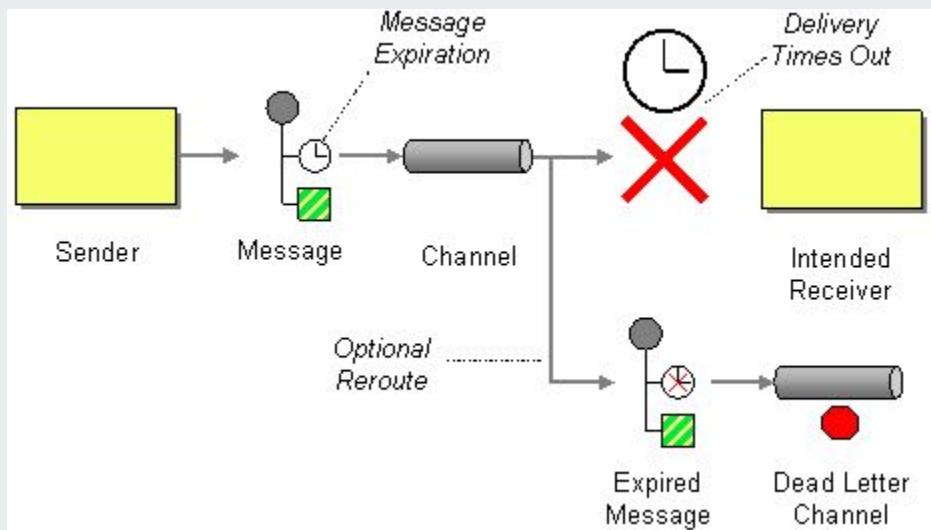
Il possède un canal d'entrée qui se divise en plusieurs canaux de sortie, un pour chaque abonné.  
Lorsqu'un événement est publié dans le canal, le canal Publish-Subscribe délivre une copie du message à chacun des canaux de sortie.  
Chaque canal de sortie n'a qu'un seul abonné, qui n'est autorisé à consommer un message qu'une seule fois.  
De cette façon, chaque abonné ne reçoit le message qu'une seule fois et les copies consommées disparaissent de leurs canaux.

## EIP : Dead Letter Channel



Lorsqu'un message ne peut pas être consommé par une application réceptrice, le message peut être envoyé vers le canal "lettre morte". Il suffit alors de superviser ce canal et d'effectuer les actions correctives si un message arrive dans ce canal.

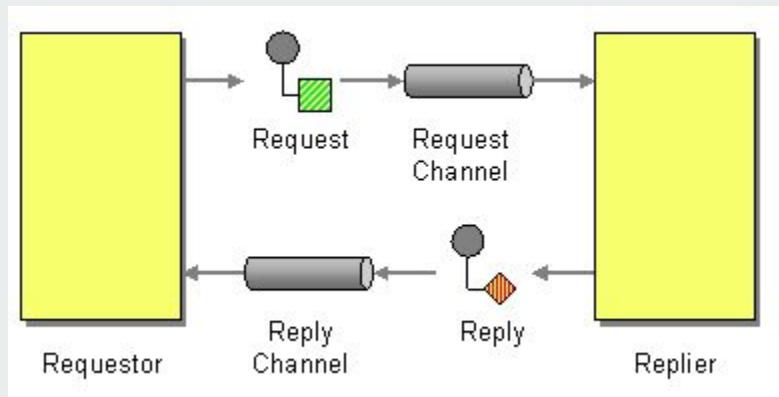
## EIP : Message Expiration



Lorsqu'un message ne peut pas être consommé par une application réceptrice dans un délai prédefini, le message peut être envoyé vers le canal "lettre morte".

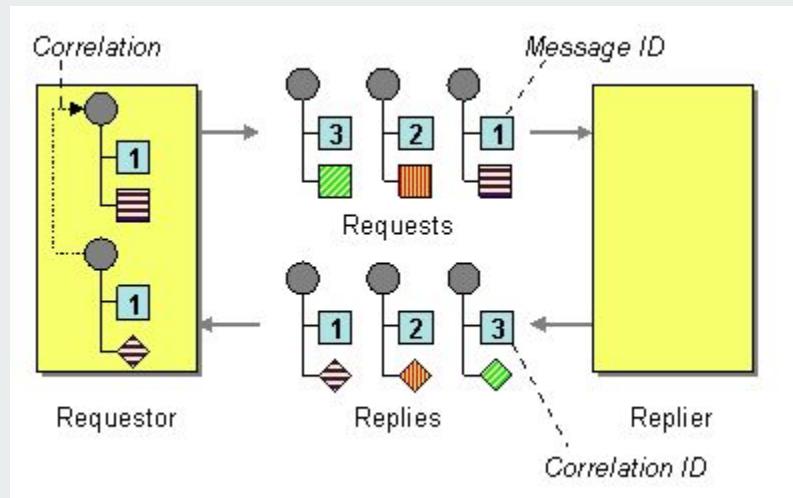
Il suffit alors de superviser ce canal et d'effectuer les actions correctives si un message arrive dans ce canal.

## EIP : Request-Reply



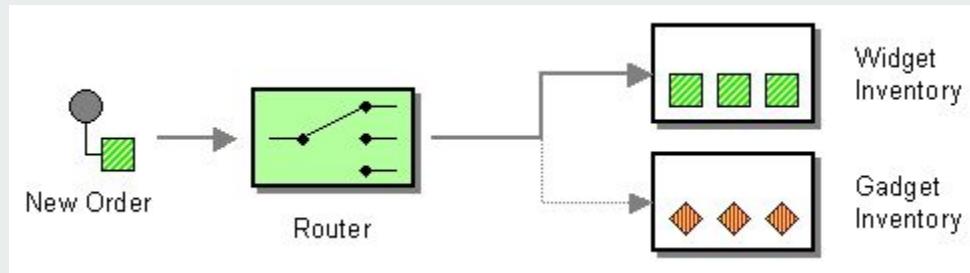
Un canal pour la requête.  
Un canal pour la réponse.  
Pas de corrélation entre la requête et la réponse.

## EIP : Correlation Identifier



Un canal pour la requête.  
Un canal pour la réponse.  
Chaque message de réponse doit contenir un identifiant de corrélation unique qui indique à quel message de requête correspond cette réponse.

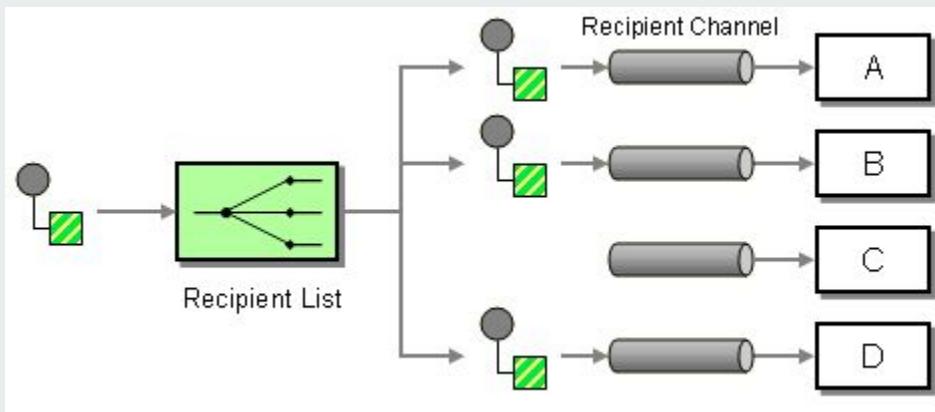
## EIP : Content-Based Router



Le routeur basé sur le contenu examine le contenu du message et l'achemine vers un canal différent en fonction des données contenues dans le message.

L'acheminement peut être basé sur un certain nombre de critères tels que l'existence de champs, des valeurs de champs spécifiques, etc.

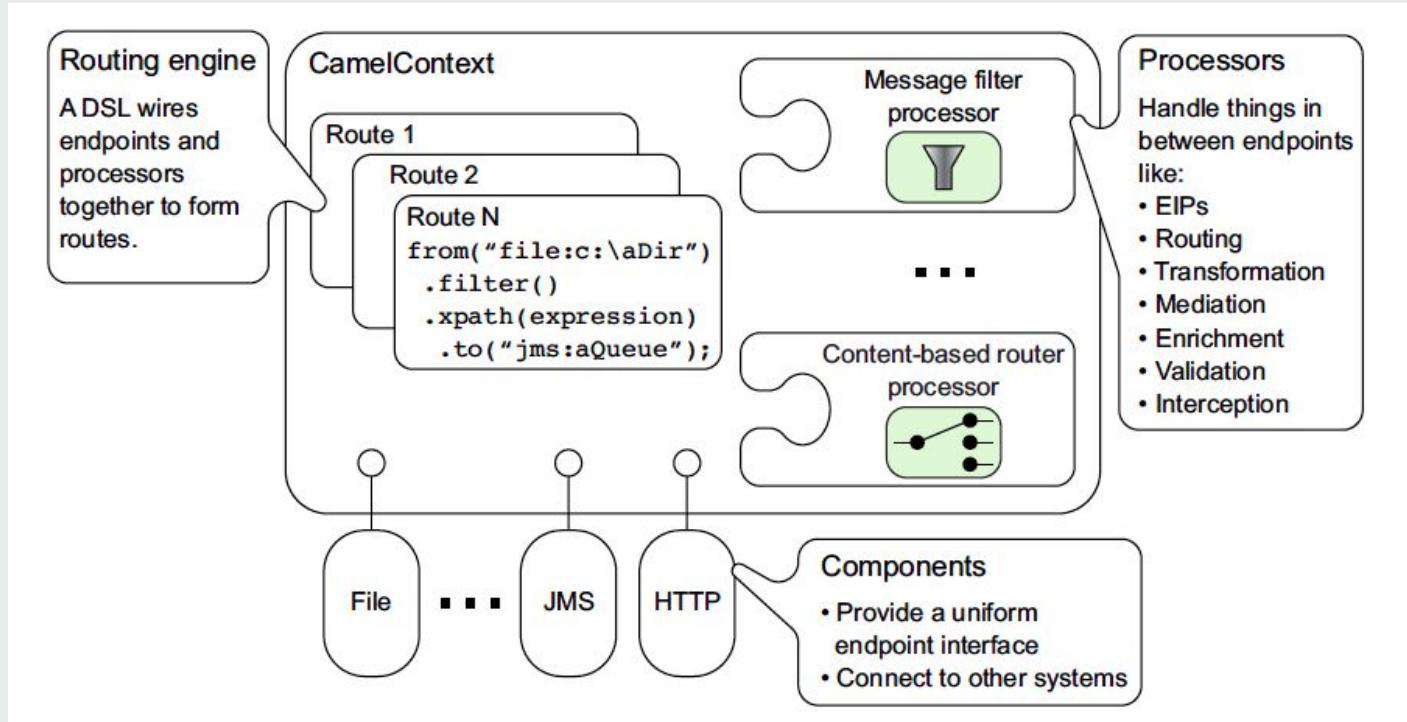
## EIP : Recipient List



Définition d'un canal pour chaque destinataire.

Inspection du message entrant pour déterminer la liste des destinataires souhaités et transmettre le message à tous les canaux associés aux destinataires de la liste.

## Focus sur Apache Camel



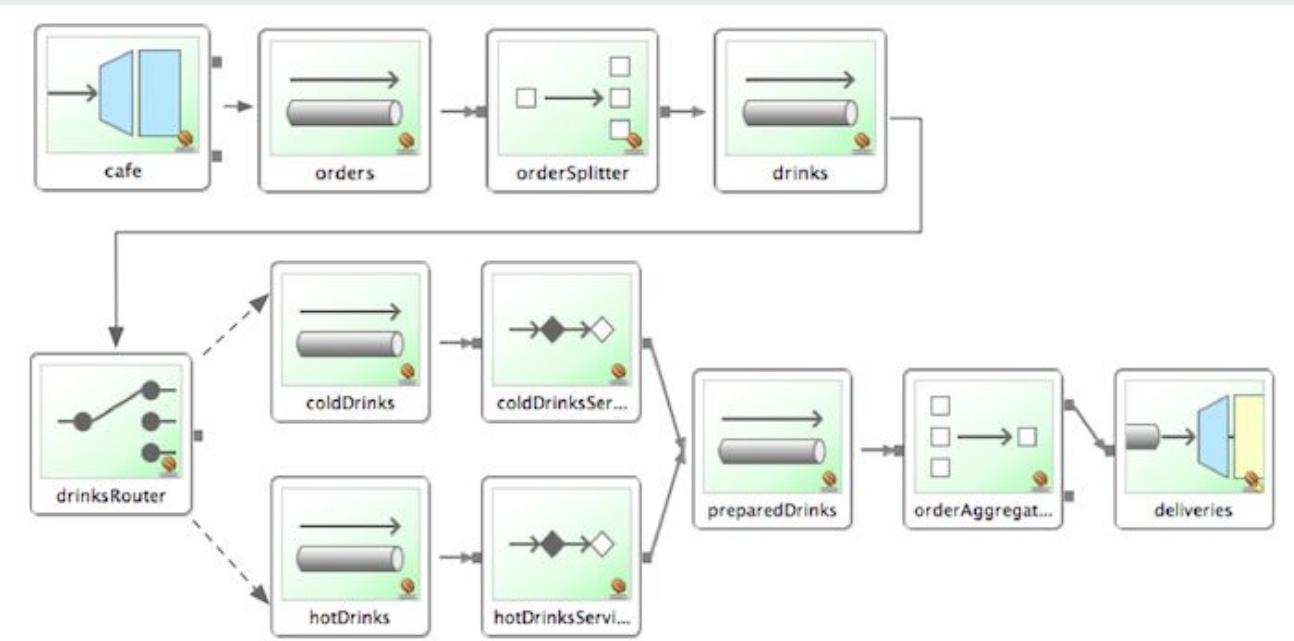
- DSL : Domain Specific Language
- EIP : Enterprise Integration Patterns

## Focus sur Apache Camel : DSL

- Exemple : Java DSL

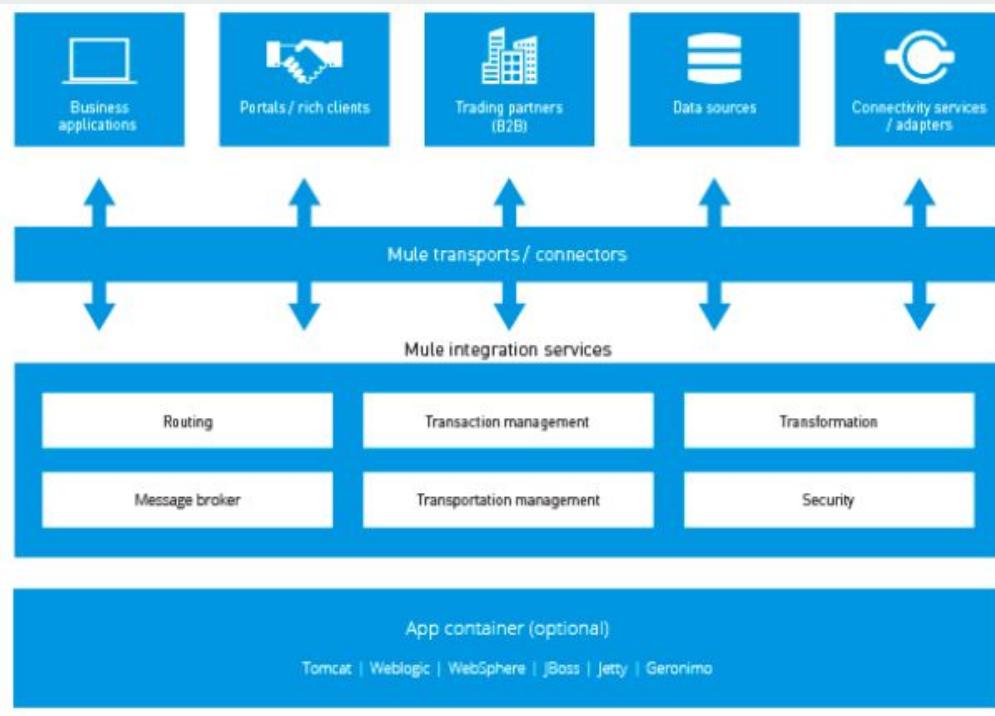
```
from("file:src/data?noop=true")
    .choice()
        .when(xpath("/person/city = 'London')")
            .to("file:target/messages/uk")
        .otherwise()
            .to("file:target/messages/others");
}
```

## Spring Integration



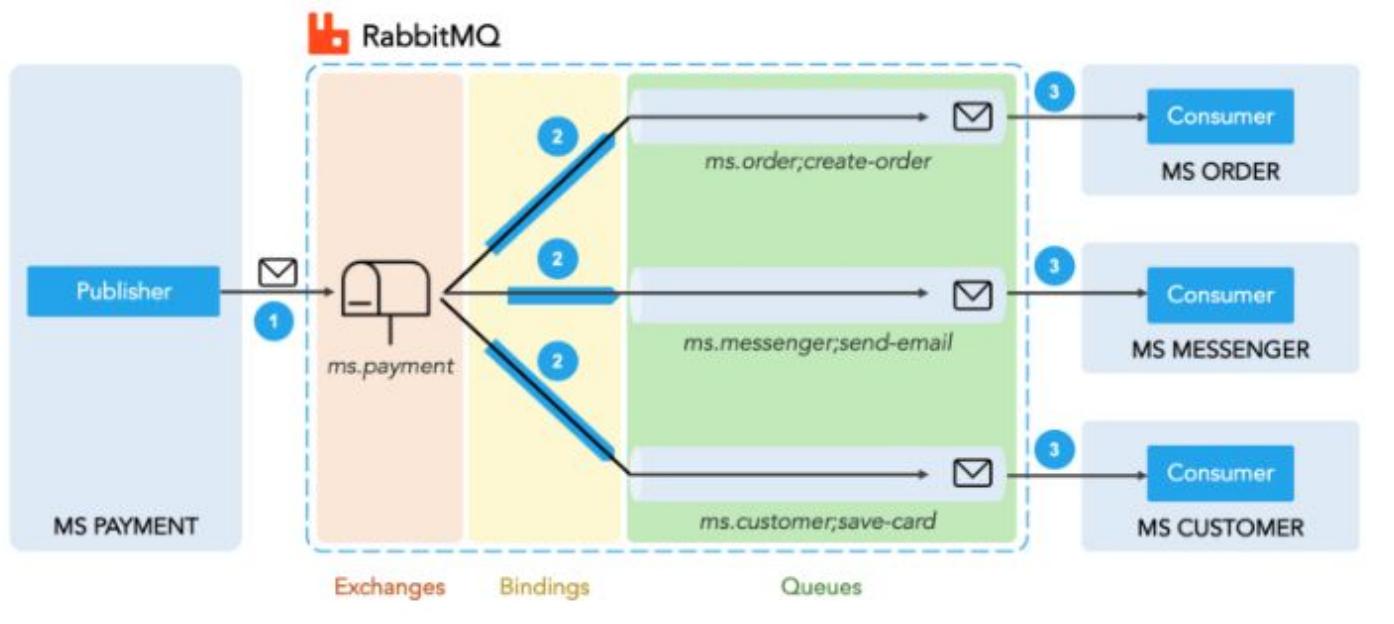
Configuration  
standalone ou avec  
Spring Boot

## Mule ESB



- éditeur : MuleSoft
- adapté pour des besoins importants d'orchestration
- surdimensionné de petits projets
- un très grand nombre de fonctionnalités :
  - connecteurs
  - éditeur graphique
- <https://www.mulesoft.com/fr/resources/esb/what-mule-esb>

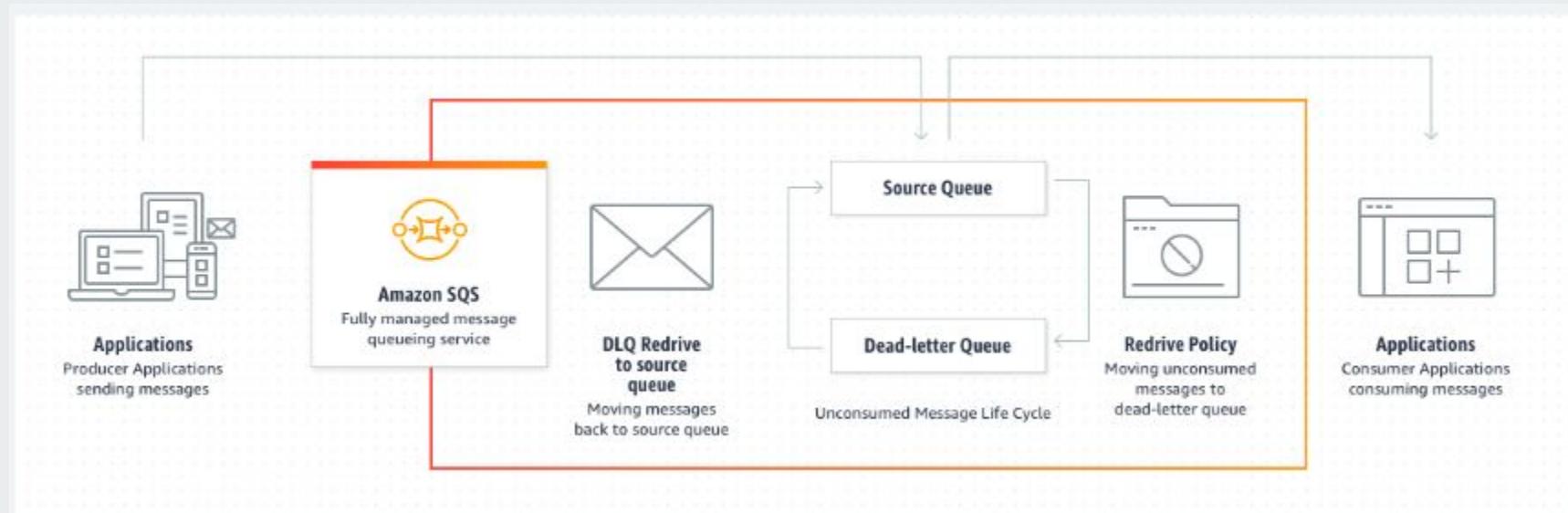
## Focus sur RabbitMQ



- message broker
- open-source
- <https://www.rabbitmq.com/>

## Focus sur AWS SQS

- Amazon Web Services : Simple Queue Service

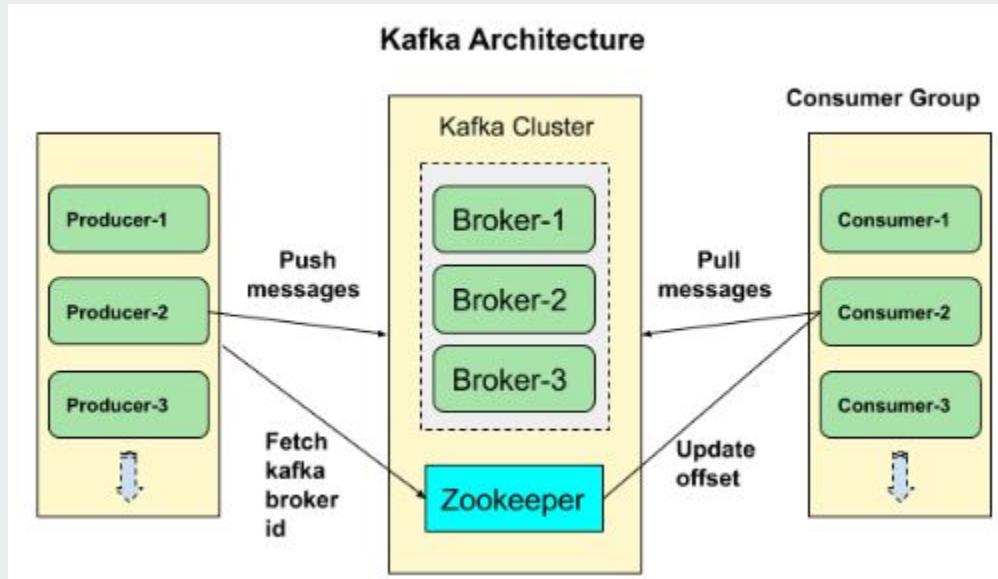


## Focus sur AWS SNS

- Amazon Web Services : Simple Notification Service



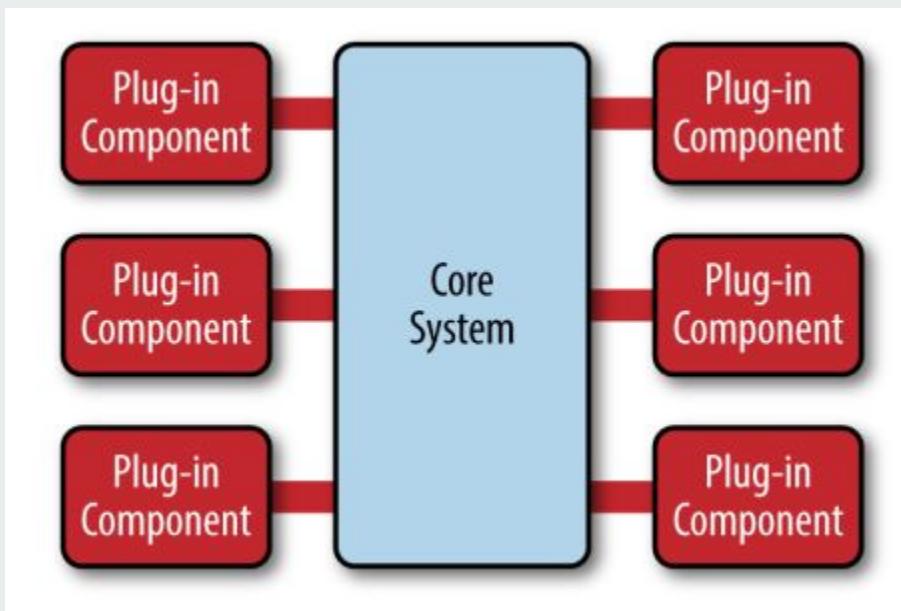
## focus sur Apache Kafka



- Apache Kafka est une plateforme distribuée de diffusion de données en continu, capable de publier, stocker, traiter et souscrire à des flux d'enregistrement en temps réel. Elle est conçue pour gérer des flux de données provenant de plusieurs sources et les fournir à plusieurs utilisateurs.
- Développé par LinkedIn comme système interne pour la gestion de ses 1 400 milliards de messages applicatifs quotidiens et cédé à la communauté Apache

<https://www.youtube.com/watch?v=9CrIA0Wasvk>

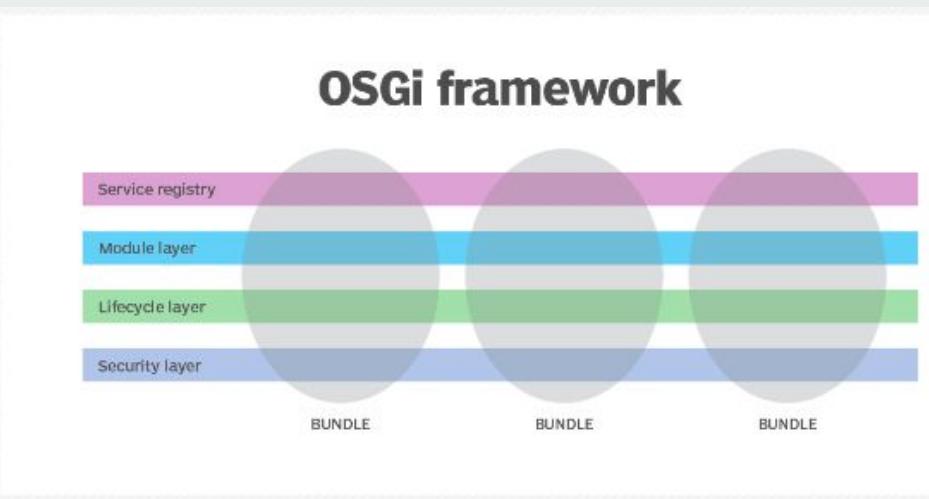
## Les technologies impliquées dans le modèle d'architecture microkernel



Rappel :

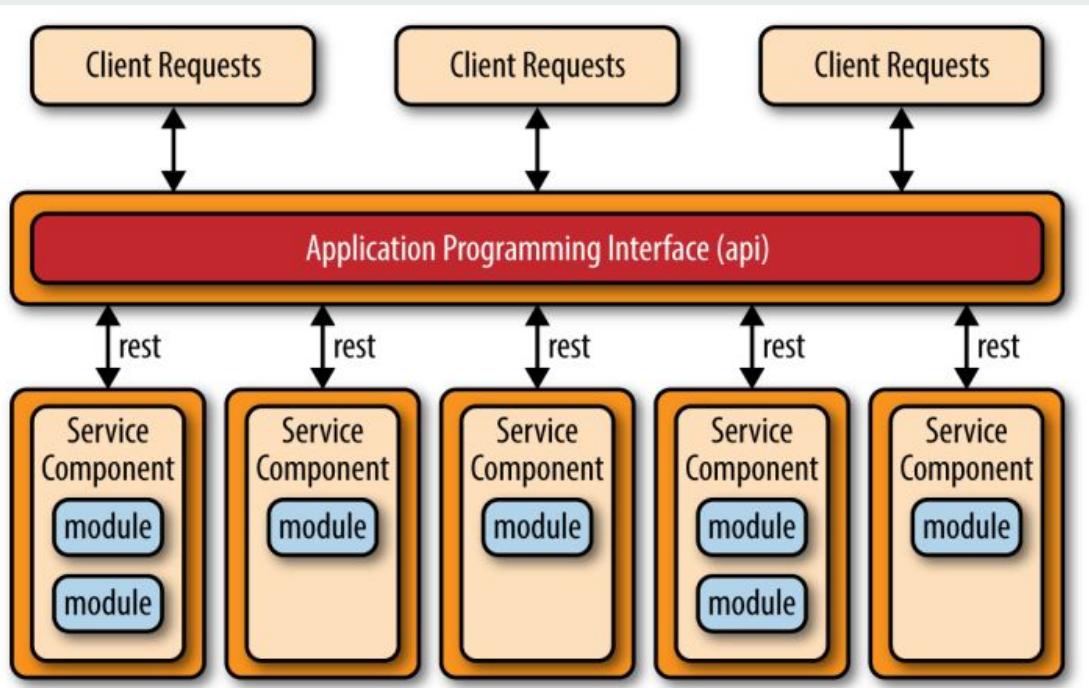
- 2 types de composants : le cœur et des plug-ins
- Idéalement, les plug-ins doivent pouvoir être déployés “à chaud”, sans redémarrage du cœur

## Les technologies impliquées dans le modèle d'architecture microkernel : OSGi



- OSGi : Open Services Gateway initiative
- Date de création : 1999
- Spécifie un modèle de gestion de cycle de vie d'une application, un répertoire (registry) de services, un environnement d'exécution et des modules.
- Basé sur Java
- Exemple d'implémentation : Apache Karaf
- Exemple d'utilisation : Eclipse IDE

## 9. Microservices et API



- REST
- HATEOAS
- Xml, Json

## REST

---

- REST : Representational State Transfer
- Date de création : début des années 2000
- Il ne s'agit ni d'un protocole, ni d'une norme, mais défini comme un style d'architecture
- On parle d'API RESTful
- En général, une requête REST s'appuie sur le protocole HTTP et échange des données entre un client et un serveur au format JSON, XML ou text. Néanmoins, REST n'impose ni le protocole ni les formats d'échanges.
- Quand REST utilise HTTP, il s'appuie sur les verbes HTTP les plus fréquents : GET, POST, PUT, DELETE
- et aussi sur les 5 grandes familles de code HTTP :  
1XX : information, 2XX : succès, 3XX : redirection,  
4XX : erreur côté client, 5XX : erreur côté serveur

## REST : exemple

<b>Endpoint</b> ←→	<code>https://apiurl.com/review/new</code>
<b>HTTP Method</b> ←→	<code>POST</code>
<b>HTTP Headers</b> ←→	<code>content-type: application/json</code> <code>accept: application/json</code> <code>authorization: Basic abase64string</code>
<b>Body</b> ←→	<pre>{     "review" : {         "title" : "Great article!",         "description" : "So easy to follow.",         "rating" : 5     } }</pre>



## HATEOAS

---

- HATEOAS : Hypertext As The Engine Of Application State
- Le principe HATEOAS introduit tout simplement des transitions possibles entre les différents états d'une même ressource, ainsi qu'entre les ressources elles-mêmes.
- Les réponses sont enrichies avec des liens hypermédias qui fournissent ces fameuses transitions.
- Considéré comme un niveau de maturité supérieur de REST (niveau 3 dans le modèle de Richardson)

## REST vs HATEOAS : 1 exemple

- Requête : demande les contrats d'un client  
GET http://api.nexworld.fr/v1/customers/1111/contracts

REST

```
HTTP/1.1 200 Success
{
  « id » : « 1111 »,
  « lastname » : « Dupond »,
  « contracts » : [
    {« id » : « 123 »},
    {« id » : « 124 »}
  ]
}
```

HATEOAS

```
HTTP/1.1 200 Success
{
  « id » : « 1111 »,
  « lastname » : « Dupond »,
  « links » : [
    {
      « rel » : « contract »,
      « method » : « GET »,
      « href » : "/contracts/123"
    },
    {
      « rel » : « contract »,
      « method » : « GET »,
      « href » : "/contracts/124"
    }
  ]
}
```

## HATEOAS : cas d'utilisation (1 / 2)

- Pagination

```
{  
  « total_items » : 166,  
  « total_pages » : 83,  
  « page » : 27,  
  « page_size » : 2,  
  « customers » : [  
    {  
      « id » : « 1234 »,  
      « lastname » : « DUPOND »,  
      « firstname » : « JEAN »  
    },  
    {  
      « id » : « 5678 »,  
      « lastname » : « DURAND »,  
      « firstname » : « ANTOINE »  
    }  
  ]  
}
```

```
  « links » : [  
    {  
      « rel » : « first »,  
      « href » : « http://api.nexworld.fr/v1/customers?page_size={page_size}&page=1 »,  
    },  
    {  
      « rel » : « next »,  
      « href » : « http://api.nexworld.fr/v1/customers?page_size={page_size}&page={page+1} »,  
    },  
    {  
      « rel » : « prev »,  
      « href » : « http://api.nexworld.fr/v1/customers?page_size={page_size}&page={page-1} »,  
    },  
    {  
      « rel » : « last »,  
      « href » : « http://api.nexworld.fr/v1/customers?page_size={page_size}&page={total_pages} »,  
    }  
  ]  
}
```

## HATEOAS : cas d'utilisation (2 / 2)

- Résolution d'erreur

Requête de mise à jour de l'adresse d'un compte client

```
PATCH http://api.nexworld.fr/v1/users/1234
{
  « address » : {
    ...
  }
}
```

La réponse contient l'action corrective

```
{
  « error_name » : « INVALID_OPERATION »,
  « error_message » : « update to an inactive account is not supported »,
  « links » : [
    {
      « href » : « http://api.nexworld.fr/v1/users/1234/activate »,
      « rel » : « activate user »,
      « method » : « POST »
    }
  ]
}
```

## HATEOAS : conclusion

- Avantages
  - Découplage client-serveur
  - Diminution des erreurs côté client
  - Logique métier portée par le serveur
  - Auto-documentation
- Inconvénients
  - Développement plus compliqué côté serveur et côté client
  - Inadapté pour certaines méthodes

## AMQP

- Le protocole AMQP (Advanced Message Queuing Protocol) est une norme open source pour les systèmes de messagerie asynchrone par réseau.
- AMQP permet l'échange de messages chiffrés et interopérables entre organisations et applications. Le protocole est utilisé pour les systèmes de messagerie client/serveur et pour la gestion de périphériques IoT.
- AMQP assure la sécurité de l'échange de messages grâce à plusieurs modes :
  - at-most-once (un seul envoi avec la possibilité qu'il soit manqué)
  - at-least-once (livraison garantie avec la possibilité de messages dupliqués)
  - exactly-once (livraison unique garantie)

## gRPC

- gRPC est un framework RPC (Remote procedure call) open source initialement développé par Google. Il utilise le protocole HTTP/2 pour le transport, Protocol Buffers comme format d'échanges de données.
- gRPC est conçu pour une faible latence et une communication à débit élevé. gRPC est idéal pour les microservices légers où l'efficacité est essentielle.
- gRPC offre une excellente prise en charge du streaming bidirectionnel. Les services gRPC peuvent envoyer des messages en temps réel sans interrogation.
- Pas de prise en charge directe par les navigateurs

## GraphQL

- GraphQL : alternative moderne à REST
- Développé par Facebook (2015) → standard ouvert ([graphql.org](https://graphql.org)).
- Langage de requête pour APIs → le client demande exactement les données dont il a besoin.
- 1 seul endpoint, pas de multiplications d'URL.
- Structure :
  - Query → lecture de données
  - 
  - Mutation → écriture / mise à jour
  - 
  - Subscription → écoute d'événements temps réel (WebSocket)

# GraphQL

- Avantages :
  - Réduit le nombre d'appels API (1 requête → plusieurs ressources).
  - Forte typage et introspection automatique (auto-doc).
  - Adapté aux architectures front multiples (Web, Mobile, IoT).
  - Compatible avec REST / BFF / microservices.
- Limites :
  - Complexité du serveur et des résolveurs.
  - Pas idéal pour le caching HTTP standard (nécessite gateway spécialisée).
  - Risques de requêtes trop coûteuses (query depth, loops).
- Exemples d'implémentations :
  - Apollo Server / Client
  - Hasura
  - AWS AppSync
  - GraphQL Yoga, Mercurius (Node.js)
- GraphQL complète REST, il ne le remplace pas systématiquement.

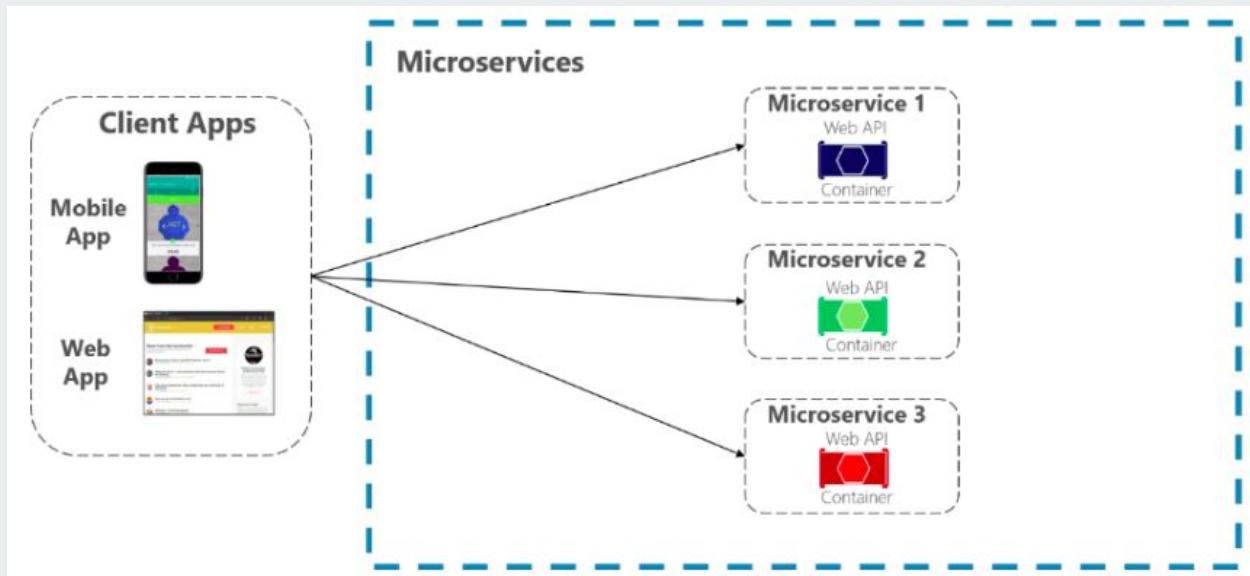
## GraphQL vs REST

Critère	REST	GraphQL
Structure	Multiples endpoints	Un seul endpoint
Format	JSON / XML	JSON typé
Appels réseau	Multiples	Un seul
Documentation	Swagger/OpenAPI	Auto-générée (Schema)
Cache	HTTP natif	Niveau client (Apollo, Relay)
Temps réel	Non natif	Oui (Subscriptions)

GraphQL simplifie le front, mais complexifie le back.  
[Comparatif détaillé](#)

## Utiliser une API gateway ou pas ?

- API Gateway = Passerelle API  
Sans passerelle API : communication directe de clients à micro-services

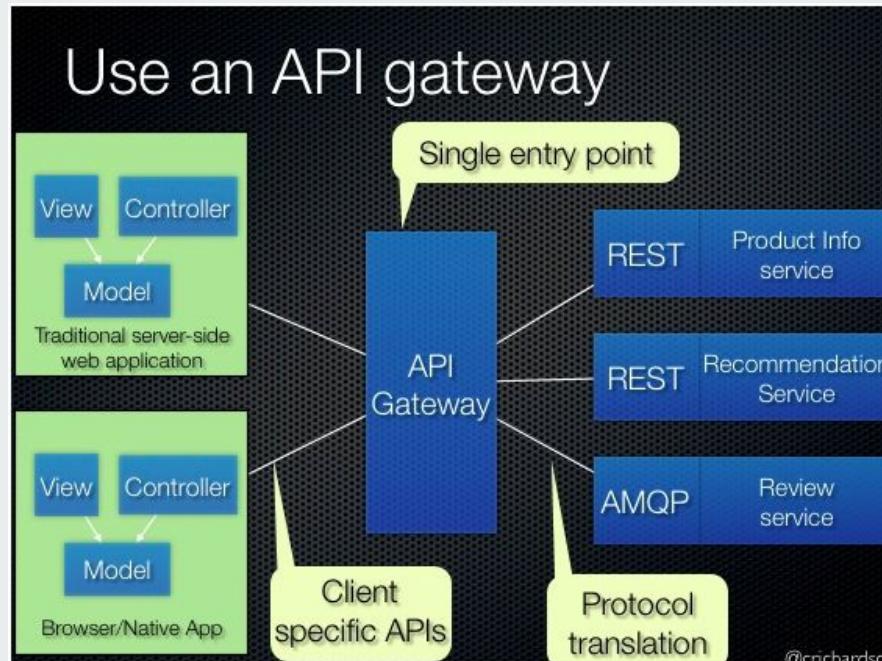


## Utiliser une API gateway ou pas ?

- Sans passerelle API : communication directe de clients à micro-services
  - peut convenir pour une petite application
- Mais présente des inconvénients :
  - pas de possibilité d'agréger des appels et des réponses
  - pas de gestion transverse pour des problématiques communes comme la sécurité
  - couplage entre le client et les micro-services

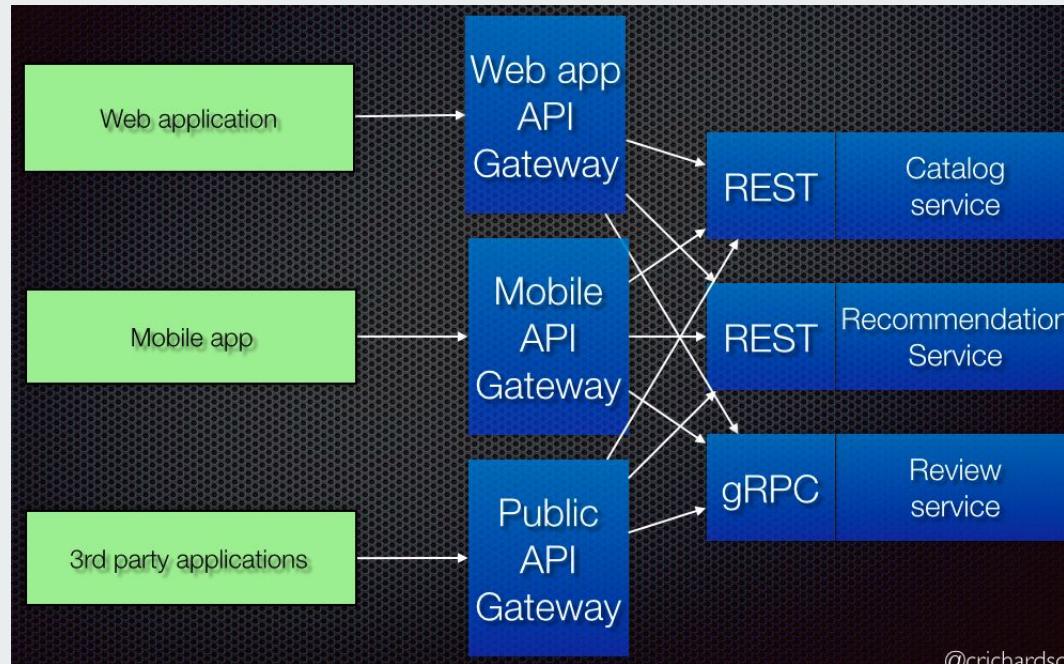
## Utiliser une API gateway ou pas ?

- Avec passerelle API : exemple 1



## Utiliser une API gateway ou pas ?

- Avec passerelle API : exemple 2



## API Gateway : définition et rôle

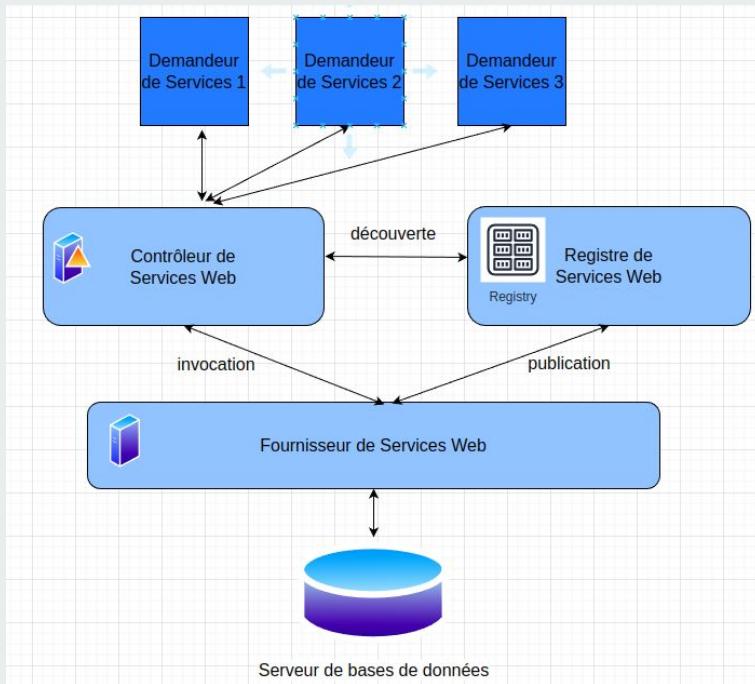
- Point d'entrée unique pour toutes les requêtes externes.
- Fonctionnalités principales :
  - Authentification et contrôle d'accès
  - Agrégation de services (composition de réponses)
  - Limitation de débit, quotas, cache
  - Monitoring, logs et traçabilité
- Permet de découpler la consommation API du détail des microservices.

## API Gateway : exemples et positionnement

Outil	Type	Points forts
<b>AWS API Gateway</b>	Cloud-native	Intégration Lambda, IAM, throttling natif
<b>Kong</b>	Open Source / Enterprise	Extensible via plugins Lua, scalable
<b>Apigee (Google)</b>	Entreprise	Politiques avancées, monétisation API
<b>Nginx Gateway</b>	Reverse proxy extensible	Simplicité, performance
<b>Tyk</b>	Open Source / Cloud	Légère, facile à automatiser
<b>Traefik</b>	Cloud-native	Parfait pour Kubernetes

AWS API Gateway : <https://www.youtube.com/watch?v=kr3GMkkzvKE> (44')

## Les technologies impliquées dans le modèle d'architecture orientée services



- SOAP : Simple Object Access Protocol. Protocole de communication entre applications qui s'appuie sur HTTP et XML.
- WSDL : Web Services Description Language. Langage de description des Web Services basé sur XML
- UDDI : Universal Description Discovery and Integration. Annuaire de services décrits en XML
- BPEL : Business Process Execution Language. Langage basé sur XML utilisé dans les moteurs d'orchestration et d'exécution des Web Services d'une architecture SOA.

---

## Les technologies impliquées dans le modèle d'architecture orientée services

```
<?xml version="1.0" encoding="UTF-8" standalone="no" ?>
<SOAP-ENV:Envelope
    SOAP-ENV:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
    xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/"
    xmlns:SOAP-ENC="http://schemas.xmlsoap.org/soap/encoding/"
    xmlns:xsi="http://www.w3.org/1999/XMLSchema-instance"
    xmlns:xsd="http://www.w3.org/1999/XMLSchema">
    <SOAP-ENV:Body>
        <ns1:doubleAnInteger
            xmlns:ns1="urn:MySoapServices">
            <param1 xsi:type="xsd:int">123</param1>
        </ns1:doubleAnInteger>
    </SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

- SOAP : Exemple de requête

## Les technologies impliquées dans le modèle d'architecture orientée services

```
<?xml version="1.0" encoding="utf-8"?>
<wsdl:definitions xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/" ...>
  <wsdl:types>
    <s:schema elementFormDefault="qualified" targetNamespace="http://tempuri.org/">
      ...
      <s:element name="GetWordListResponse">
        <s:complexType>
          <s:sequence>
            <s:element minOccurs="0" maxOccurs="1" name="GetWordListResult" type="Ins:ArrayOfString" />
          </s:sequence>
        </s:complexType>
      </s:element>
    </s:schema>
  </wsdl:types>
  <wsdl:message name="GetWordListSoapIn">
    <wsdl:part name="parameters" element="Ins:GetWordList" />
  </wsdl:message>
  <wsdl:message name="GetWordListSoapOut">
    <wsdl:part name="parameters" element="Ins:GetWordListResponse" />
  </wsdl:message>
  <wsdl:portType name="AutoWebServiceSoap">
    <wsdl:operation name="GetWordList">
      <wsdl:input message="Ins:GetWordListSoapIn" />
      <wsdl:output message="Ins:GetWordListSoapOut" />
    </wsdl:operation>
  </wsdl:portType>
  <wsdl:binding name="AutoWebServiceSoap" type="tns:AutoWebServiceSoap">
    <soap:binding transport="http://schemas.xmlsoap.org/soap/http" />
    <wsdl:operation name="GetWordList">
      <soap:operation soapAction="http://tempuri.org/GetWordList" style="document" />
      <wsdl:input>
        <soap:body use="literal" />
      </wsdl:input>
      <wsdl:output>
        <soap:body use="literal" />
      </wsdl:output>
    </wsdl:operation>
  </wsdl:binding>
  <wsdl:service name="AutoWebService">
    <wsdl:port name="AutoWebServiceSoap" binding="tns:AutoWebServiceSoap">
      <soap:address location="http://www.vietnamofficeexpress.com/AutoWebService.asmx" />
    </wsdl:port>
  </wsdl:service>
</wsdl:definitions>
```

- Exemple de WSDL

## Quiz 4a

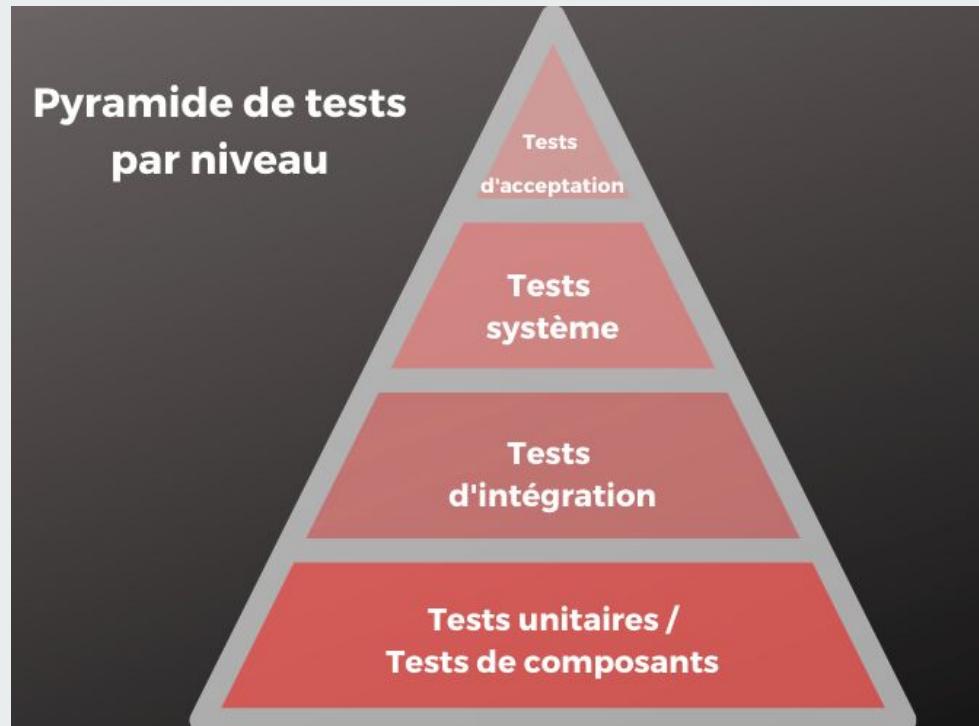


[https://docs.google.com/forms/d/e/1FAIpQLScpUBkQPYXsGG4IyJWMV2UglyamaF\\_H5r4mHk1\\_Kv3-SY\\_NAg/viewform?usp=dialog](https://docs.google.com/forms/d/e/1FAIpQLScpUBkQPYXsGG4IyJWMV2UglyamaF_H5r4mHk1_Kv3-SY_NAg/viewform?usp=dialog)



## 10. Tests logiciels et qualité continue

## Les différents types de tests



## Les différents types de tests

- Tests unitaires :

De très bas niveau, ils consistent à tester les méthodes et fonctions individuelles des classes, des composants ou des modules utilisés par le logiciel.

Les tests unitaires peuvent être automatisés et exécutés très rapidement par un serveur d'intégration continue.

Outils : un très grand nombre d'outils :

[https://en.wikipedia.org/wiki/List\\_of\\_unit\\_testing\\_frameworks](https://en.wikipedia.org/wiki/List_of_unit_testing_frameworks)

## Les différents types de tests

- Tests d'intégration :

Les tests d'intégration vérifient que les différents modules ou services utilisés par votre application fonctionnent bien ensemble. Par exemple, ils peuvent tester l'interaction avec la base de données ou s'assurer que les microservices fonctionnent ensemble comme prévu.

Ces types de tests sont plus coûteux à exécuter, car ils nécessitent que plusieurs parties de l'application soient fonctionnelles.

Outils : certains outils de tests unitaires peuvent aussi être utilisés pour les tests d'intégration.

Quelques outils spécialisés : Fitnesse, VectorCast, Citrus

## Les différents types de tests

- Tests fonctionnels :

Les tests fonctionnels se concentrent sur les exigences métier d'une application. Ils vérifient uniquement la sortie d'une action et non les états intermédiaires du système lors de l'exécution de cette action.

Il y a parfois une certaine confusion entre les tests d'intégration et les tests fonctionnels, car ils nécessitent tous les deux plusieurs composants pour interagir. La différence réside dans le fait qu'un test d'intégration peut simplement vérifier que vous pouvez interroger la base de données, tandis qu'un test fonctionnel s'attend à obtenir une valeur spécifique de la base de données, telle que définie par les exigences du produit.

Outils : mêmes outils que les tests d'intégration  
Quelques outils spécialisés : Selenium, SoapUi

## Les différents types de tests

- Tests de bout en bout :

Les tests de bout en bout reproduisent le comportement d'un utilisateur avec le logiciel dans un environnement applicatif complet. Ils vérifient que les différents flux d'utilisateurs fonctionnent comme prévu et peuvent être aussi simples que le chargement d'une page Web ou la connexion. Des scénarios beaucoup plus complexes peuvent aussi vérifier les notifications par e-mail, les paiements en ligne, etc.

Les tests de bout en bout sont très utiles, mais ils sont coûteux à réaliser et peuvent être difficiles à gérer lorsqu'ils sont automatisés. Il est recommandé d'avoir quelques tests clés de bout en bout et de s'appuyer davantage sur des tests de niveau inférieur (tests unitaires et d'intégration) pour être en mesure d'identifier rapidement les régressions.

Outils : exemple : Selenium + Bitbar (Smartbear) ou [BrowserStack](#)

## Les différents types de tests

- Tests d'acceptation :

Les tests d'acceptation sont des tests formels exécutés pour vérifier si un système répond à ses exigences métier. Ils nécessitent que l'application soit entièrement opérationnelle et se concentrent sur la simulation du comportement des utilisateurs.

Ils peuvent aussi aller plus loin, et mesurer la performance du système et rejeter les changements si certains objectifs ne sont pas atteints.

## Les différents types de tests

- *Smoke tests :*

Les *smoke tests* sont des tests simples qui vérifient le fonctionnement de base d'une application. Ils sont conçus pour être rapides à exécuter, et leur but est de vous donner l'assurance que les caractéristiques principales de votre système fonctionnent comme prévu.

Les « *smoke tests* » peuvent être utiles juste après la création d'un build afin de décider si vous pouvez exécuter des tests plus coûteux. Ils peuvent également être utiles après un déploiement afin de vous assurer que l'application s'exécute correctement dans l'environnement nouvellement déployé.

## Les différents types de tests : tests liés à la performance du logiciel

- Test de charge :  
Test au cours duquel on va simuler un nombre d'utilisateurs virtuels prédéfinis, afin de valider l'application pour une charge attendue d'utilisateurs. Ce type de test permet de mettre en évidence les points sensibles et critiques de l'architecture technique. Il permet en outre de mesurer le dimensionnement des serveurs, de la bande passante nécessaire sur le réseau, etc.
- Test de performance :  
Test au cours duquel on va mesurer les performances de l'application soumise à une charge d'utilisateurs. Les informations recueillies concernent les temps de réponse utilisateurs, les temps de réponse réseau et les temps de traitement d'une requête sur le(s) serveur(s).  
La nuance avec le Test de charge réside dans le fait qu'on ne cherche pas ici à valider les performances pour la charge attendue en production, mais plutôt à vérifier les performances intrinsèques à différents niveaux de charge d'utilisateurs.

## Les différents types de tests : tests liés à la performance du logiciel

- Test de dégradations des transactions :  
Test au cours duquel on ne va simuler que l'activité transactionnelle d'un seul scénario fonctionnel parmi tous les scénarios du périmètre des tests, de manière à déterminer quelle charge le système est capable de supporter pour chaque scénario fonctionnel et d'isoler éventuellement les transactions qui dégradent le plus l'ensemble du système. Ce test sert à déterminer les points de contention générés par chaque scénario fonctionnel.
- Test de stress :  
Test au cours duquel on va simuler l'activité maximale attendue tous scénarios fonctionnels confondus en heure de pointe de l'application, pour voir comment le système réagit au maximum de l'activité attendue des utilisateurs.  
Dans le cadre de ces tests, il est possible de pousser le stress jusqu'à simuler des défaillances systèmes ou applicatives afin d'effectuer des tests de récupération sur incident (Fail-over) ou pour vérifier le niveau de service en cas de défaillance.

## Les différents types de tests : tests liés à la performance du logiciel

- Test de robustesse, d'endurance, de fiabilité :  
Tests au cours desquels on va simuler une charge importante d'utilisateurs sur une durée relativement longue, pour voir si le système testé est capable de supporter une activité intense sur une longue période sans dégradations des performances et des ressources applicatives ou système. Le résultat est satisfaisant lorsque l'application a supporté une charge supérieure à la moitié de la capacité maximale du système, ou lorsque l'application a supporté l'activité d'une journée ou plusieurs jours/mois/années, pendant 8 à 10 heures, sans dégradation de performance (temps, erreurs), ni perte de ressources systèmes.
- Test de capacité, test de montée en charge :  
Test au cours duquel on va simuler un nombre d'utilisateurs sans cesse croissant de manière à déterminer quelle charge limite le système est capable de supporter. L'objectif du test est de déterminer la capacité maximale de l'ensemble système-applicatif dans une optique prévisionnelle (cf. capacity planning, gestion de la capacité en français) ;

## Les différents types de tests : tests liés à la performance du logiciel

- Test aux limites :  
Test au cours duquel on va simuler en général une activité bien supérieure à l'activité normale, pour voir comment le système réagit aux limites du modèle d'usage de l'application. Proche du test de capacité, il ne recouvre pas seulement l'augmentation d'un nombre d'utilisateurs simultanés qui se limite ici à un pourcentage en principe prédéfini, mais aussi les possibilités d'augmentation du nombre de processus métier réalisés dans une plage de temps ou en simultané, en jouant sur les cadences d'exécutions, les temps d'attente, mais aussi les configurations de la plateforme de test dans le cadre d'architectures redondées (Crash Tests).
- Test de résilience :  
Test au cours duquel on va simuler une activité réelle tout en simulant des pannes en environnement réel et vérifier que le système continue à fonctionner. Proche du test de robustesse, il permet de s'assurer de la résilience des applications et de l'infrastructure sous-jacente. Pour provoquer les pannes pendant le test, on peut utiliser des outils qui mettent aléatoirement hors service des instances dans l'environnement de test.

## Les différents types de tests : tests liés à la performance du logiciel

- Les outils (liste non exhaustive) :
  - Apache JMeter
  - LoadNinja (Smartbear)
  - NeoLoad
  - LoadUI Pro (Soap, Rest)
  - Silk Performer
  - Gatling
  - LoadRunner (Micro Focus)
  - Chaos Monkey

## Les différents types de tests : autres types de test

---

- Tests de Benchmark (comparaisons de logiciel, matériels, architectures...),
- Tests de Volumétrie des données,
- Tests de Sécurité
- Tests exploratoires

Video : Je délègue tous mes tests à une IA :

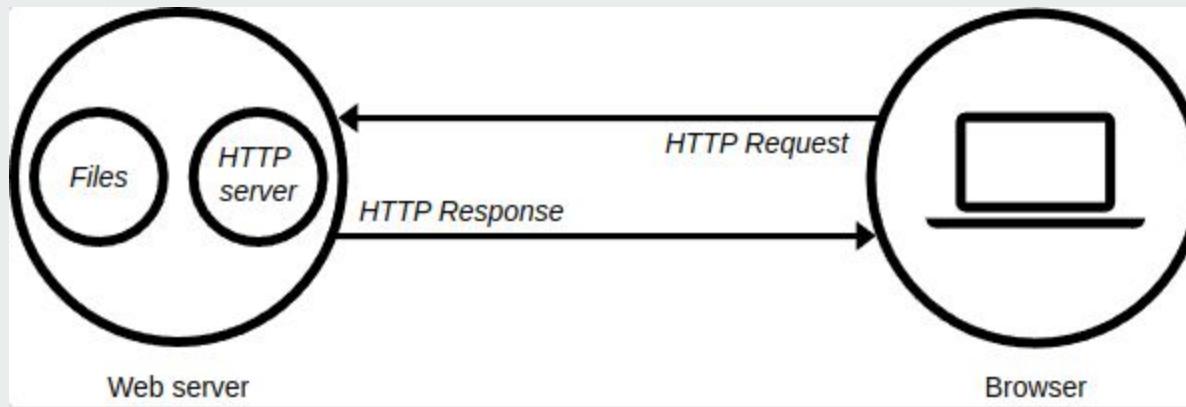
[https://www.youtube.com/watch?v=\\_TMa1JWrDHw](https://www.youtube.com/watch?v=_TMa1JWrDHw)



## 11. Intégration, déploiement et observabilité

## Les environnements d'exécution “serveur”

- Les serveurs http
  - Apache Http Server
  - Nginx
  - Microsoft IIS
  - Lighttpd



## Les environnements d'exécution “serveur”

- Les serveurs d'applications Java
  - Apache Tomcat
  - JBoss
  - GlassFish
  - IBM Websphere Application Server
- Environnement JavaScript côté serveur : Node.js

## CI / CD

---

- Intégration continue (CI) : ensemble de pratiques consistant à tester de manière automatisée chaque révision de code avant de le déployer en production.
- Déploiement continu (CD) : mise à jour automatique en préprod/prod.

## CI / CD

- Outils :
  - Jenkins
  - Bamboo (Atlassian)
  - GitHub Actions
  - Gitlab CI
  - TeamCity (Jetbrains)
  - Travis CI
  - ArgoCD
- Pratiques modernes :
  - Feature flags
  - blue/green deploy
  - canary release.

## CI / CD

Exemple Jenkins :

▷ Lancer un build

⌚ Voir la configuration

🔍 Full Stage View

⚡ SonarQube

🌐 Open Blue Ocean

(Bitbucket)

Pipeline Syntax

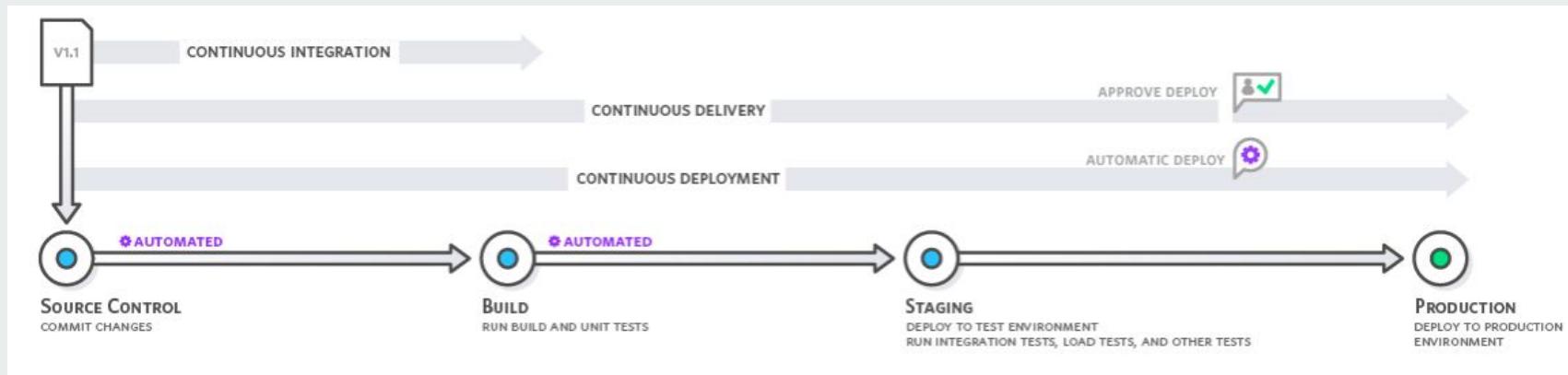
**Stage View**

	Declarative: Checkout SCM	Build	Test	[DEV] Deploy	[PREPROD] Deploy	[PROD] Deploy	Sonar	Declarative: Post Actions
Average stage times: (Average full run time: ~3min 56s)	19s	42s	54s	12s	0ms	0ms	1min 57s	96ms
#904 Nov 16 16:48 1 commit	21s	43s	46s	9s			1min 50s	78ms
#903 Nov 15 15:11 1 commit	21s	42s	46s	10s			1min 56s	83ms
#902 Nov 10 09:55 2 commits	16s	52s	1min 27s	19s			<b>2min 15s</b> failed	128ms

Résultats des tests

Passed | Skipped | Failed

## CI / CD



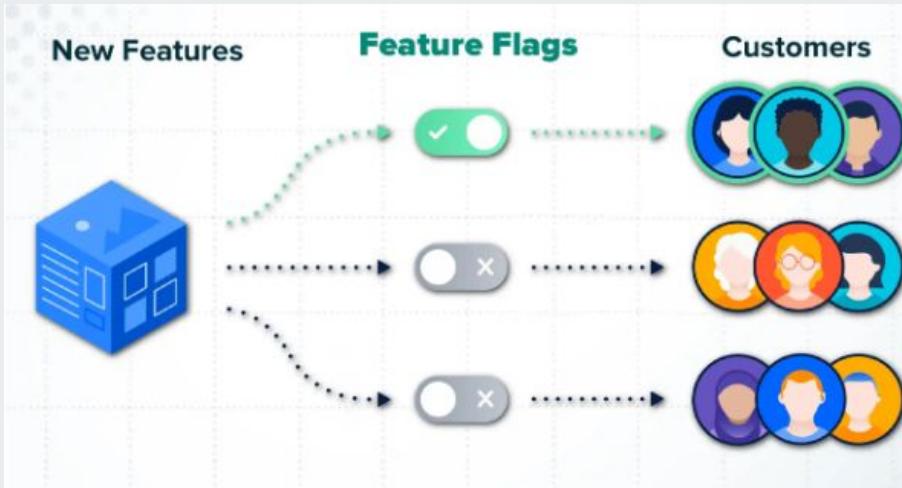
Grâce à la livraison continue, chaque modification de code est appliquée, testée puis envoyée vers un environnement de test ou de préparation hors production.

Plusieurs procédures de test peuvent avoir lieu en parallèle avant un déploiement de production. La différence entre la livraison continue et le déploiement continu réside dans la présence d'une approbation manuelle pour mettre à jour et produire.

Avec le déploiement continu, la mise en production se fait automatiquement, sans approbation explicite.

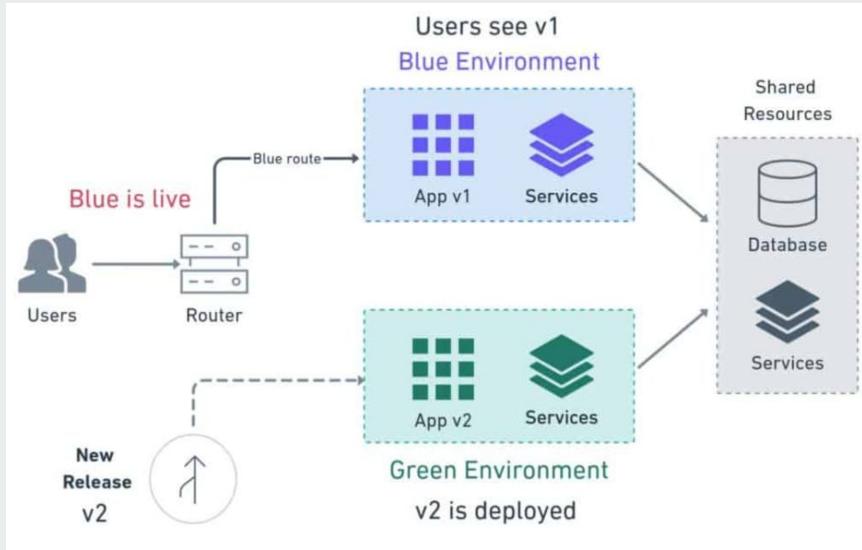
## Focus : feature flag

- Un feature flag (encore appelés « feature toggle », « feature switch » ou « feature flagging ») permet l'exécution conditionnelle du code d'un site ou d'une application pour activer ou désactiver une fonctionnalité.



## Focus : blue/green deploy

- approche de déploiement qui consiste à avoir deux environnements de production en parallèle, l'un étant actif (environnement « bleu ») et l'autre étant prêt à recevoir les nouvelles versions de l'application (environnement « vert »).



## Focus : canary release

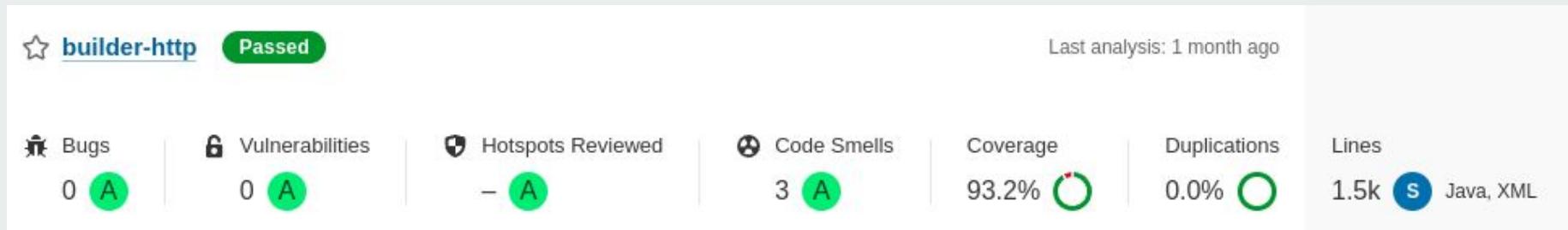
- déploiement des modifications de manière progressive à un nombre restreint d'utilisateurs.
- exemple : <https://martinfowler.com/bliki/CanaryRelease.html>

## Inspection Continue

- Augmentation de la qualité du code par analyse statique
- SonarQube
  - métriques
  - bugs
  - analyse sécurité
  - % de couverture du code par les tests unitaires
  - 29 langages
  - intégration IDE

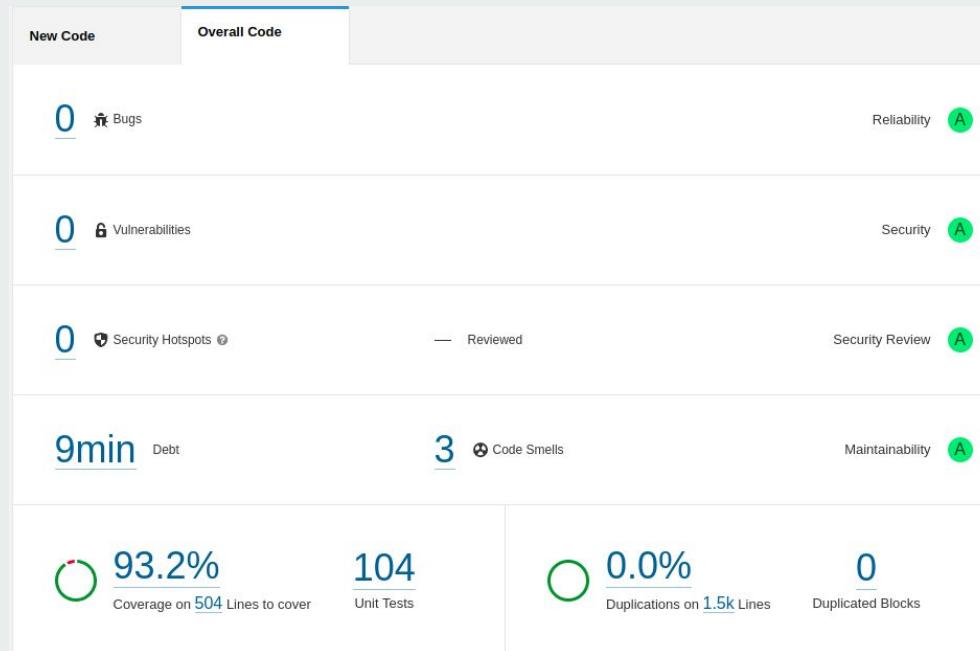
## Inspection Continue

- [SonarQube](#) : exemples



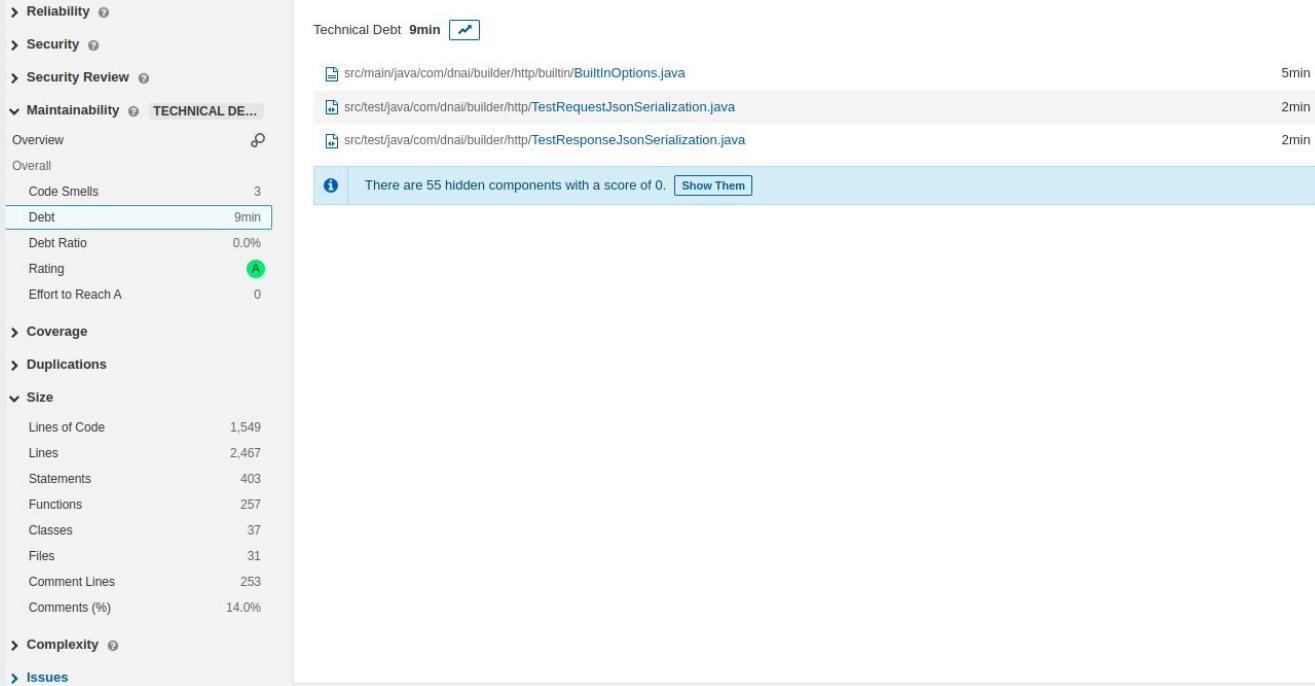
## Inspection Continue

- SonarQube : exemples



# Inspection Continue

- SonarQube : exemples



The screenshot shows the SonarQube interface for a Java project. On the left, a sidebar navigation includes Reliability, Security, Security Review, Maintainability (selected), Coverage, Duplications, and Size. The Maintainability section shows a Debt Ratio of 0.0% and an Effort to Reach A of 0. The main panel displays a 'Technical Debt' section with a 9min estimate and three specific issues listed: src/main/java/com/dnai/builder/http/BuiltinOptions.java (5min), src/test/java/com/dnai/builder/http/TestRequestJsonSerialization.java (2min), and src/test/java/com/dnai/builder/http/TestResponseJsonSerialization.java (2min). A message at the bottom indicates 55 hidden components with a score of 0, with a 'Show Them' button.

Category	Value
Code Smells	3
Debt	9min
Debt Ratio	0.0%
Rating	A
Effort to Reach A	0
Lines of Code	1,549
Lines	2,467
Statements	403
Functions	257
Classes	37
Files	31
Comment Lines	253
Comments (%)	14.0%

## Infrastructure as Code et Observabilité

---

- IaC : Terraform, Ansible, Pulumi → décrire l'infrastructure comme du code.  
exemple Terraform :  
[https://registry.terraform.io/providers/hashicorp/aws/latest/docs/resources/s3\\_bucket#example-usage](https://registry.terraform.io/providers/hashicorp/aws/latest/docs/resources/s3_bucket#example-usage)
- Observabilité à 3 piliers :
  - Logs → Stack ELK (Elastic, Logstash, Kibana)
  - Metrics → Prometheus, Grafana.
  - Traces → OpenTelemetry, Jaeger
- Objectif : comprendre le pourquoi d'un incident, pas seulement le quoi.

## logging

- Utilisation par environnement :
- En phase de développement, les logs sont complémentaires au debugger et permettent de comprendre le fonctionnement d'une application en suivant pas à pas le fil d'exécution de différentes fonctions critiques.
- En phase d'intégration et de recette, ils permettent de faciliter l'analyse des anomalies remontées.
- Enfin, en phase d'exploitation, les logs peuvent permettre de diagnostiquer des problèmes de prod remontés par le service utilisateur ou l'équipe MCO. De manière proactive, il est également possible de configurer des alertes sur des patterns d'erreur détectés dans les logs.

## logging : bonnes pratiques

---

- Centraliser les logs
- Normaliser les données de logs
- Adapter le niveau de log applicatif en fonction de l'environnement :  
En production et pré-production, il est recommandé de positionner le niveau de log applicatif à INFO. Ne tracer que les WARN et ERROR masquerait les logs précédents pouvant être utiles à l'interprétation de l'erreur. Les logs INFO trop verbeux sont à abaisser en DEBUG.
- Ne pas loguer des données sensibles :
  - Les mots de passe
  - Informations nominatives : nom, prénom, nom de naissance, numéro de sécurité sociale
  - Coordonnées bancaires : IBAN, RIB, numéro de carte bancaire
  - Informations de localisation : adresse postale, adresse IP, e-mail
  - Données de santé, génétiques et biométriques
- Changement à chaud du niveau de logs
- Maîtriser la volumétrie des logs

## logging : quelques outils

---

- Graylog (centralisation)
- [Logstash](#)
- [Splunk](#)

## monitoring : les éléments à surveiller

---

- les performances, temps de réponse des différentes ressources du serveur ;
- l'intégrité, vérification que le contenu des pages web ne change pas ;
- la disponibilité : vérifier que l'application assure l'intégralité de ses fonctionnalités (UP/DOWN).
- la charge CPU et Mémoire
- le nombre de connexions simultanées (TCP, UDP, applicative...)
- les erreurs du serveur
- la simulation d'une interaction avec l'application
- la charge du réseau (QOS, latency, ping)
- les tentatives de connexions bloquées par le pare-feu (détection d'un Nmap par exemple).

## monitoring : les éléments à surveiller

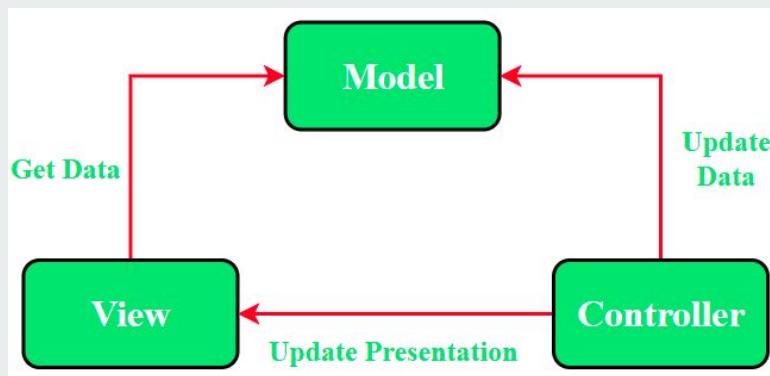
---

- Zabbix : système
- Nagios : système et réseau
- [Uptime Robot](#) : disponibilité
- [Aws CloudWatch](#)
- [Pingdom](#) : disponibilité
- Dynatrace : APM



## 12. Principes et patterns architecturaux

## Le pattern architectural MVC (1 / 2)

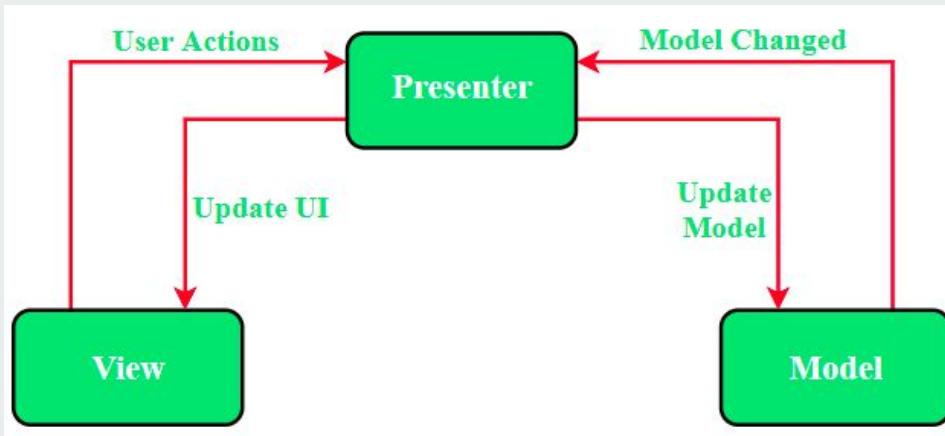


- MVC : Model - Vue - Contrôleur
- création : fin des années 1970
- Un modèle (Model) contient les données à afficher.
- Une vue (View) contient la présentation de l'interface graphique.
- Un contrôleur (Controller) contient la logique concernant les actions effectuées par l'utilisateur.

## Le pattern architectural MVC (2 / 2)

- Les problèmes rencontrés avec MVC :
  - La vue et le modèle sont étroitement couplés.  
Par conséquent, les exigences en matière de fonctionnalités de la vue peuvent facilement se propager dans le modèle.
  - La vue est monolithique et est généralement étroitement couplée au framework de présentation. Ainsi, les tests unitaires de la vue deviennent difficiles
- Vidéo : Architecture - Patron MVC :  
<https://www.youtube.com/watch?v=NhFTswve-8o>
- Variantes de MVC
  - MVP : Model - View - Presenter

## Le pattern architectural MVP (1 / 2)

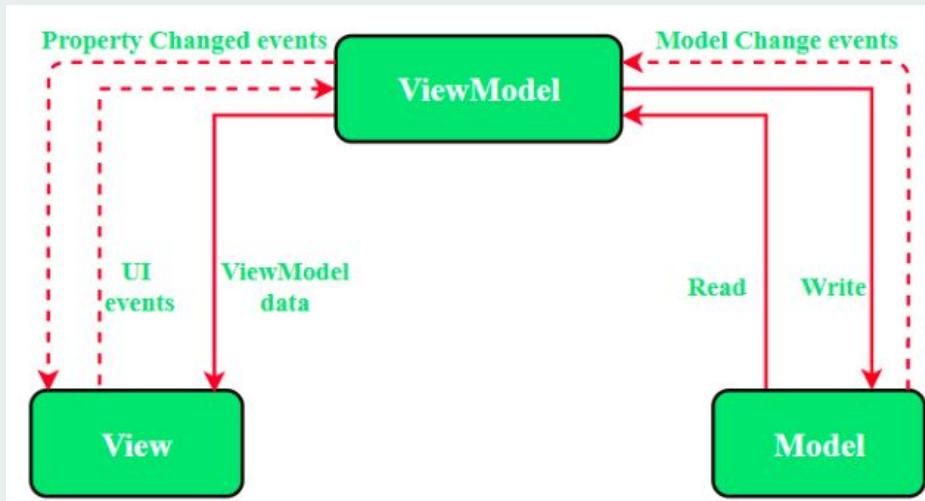


- MVP : Model - Vue - Presenter
- Model : Il est responsable de la gestion de la logique du domaine (règles métier du monde réel) et de la communication avec la base de données et les couches réseau.
- View : couche UI (interface utilisateur). Il fournit la visualisation des données et garde une trace de l'action de l'utilisateur afin d'informer le présentateur.
- Presenter : il fournit les données du modèle et applique la logique de l'interface utilisateur pour décider quoi afficher.

## Le pattern architectural MVP (2 / 2)

- Les inconvénients de MVP :
  - pas de databinding .
  - pas de réutilisation des vues ce qui crée des redondances de code de présentation.
- Variantes de MVP
  - MVVM : Model - View - View - Model

## Le pattern architectural MVVM (1 / 2)



- MVVM : Model - View - View - Model
- Model : Il est responsable de la gestion de la logique du domaine (règles métier du monde réel) et de la communication avec la base de données et les couches réseau.
- View : Le but de cette couche est d'informer le ViewModel de l'action de l'utilisateur. Cette couche observe le ViewModel et ne contient aucun type de logique d'application.
- ViewModel : il expose les flux de données qui sont pertinents pour la vue. De plus, il sert de lien entre le modèle et la vue.

## Le pattern architectural MVVM (2 / 2)

- Les problèmes résolus par MVVM :
  - databinding
  - réutilisation possible des vues
- Les inconvénients de MVVM :
  - complexité de mise en oeuvre (mais surtout de débogage dans certains cas)
  - consommation importante de ressources dans certains cas
- Vidéo : Architecture - Patron MVVM :  
<https://www.youtube.com/watch?v=UzGYdZSuL7Q>

## Le pattern architectural CQRS (1 / 3)

- CQRS : Command Query Responsability Segregation
- Principe : séparer les fonctions de mises à jour des données et les fonctions de lecture de ces données

## Le pattern architectural CQRS (2 / 3)

- Cas classique :

MyLibrary
POST createBook(book)
GET getBooks(): List<Book>
GET getBook(id): Book
PUT updateBook(id)
DELETE deleteBook(id)

Tous les points d'entrée nécessaires pour lire et mettre à jour les données.

- Avec CQRS :

MyLibraryCommand
POST createBook(book)
PUT updateBook(id)
DELETE deleteBook(id)

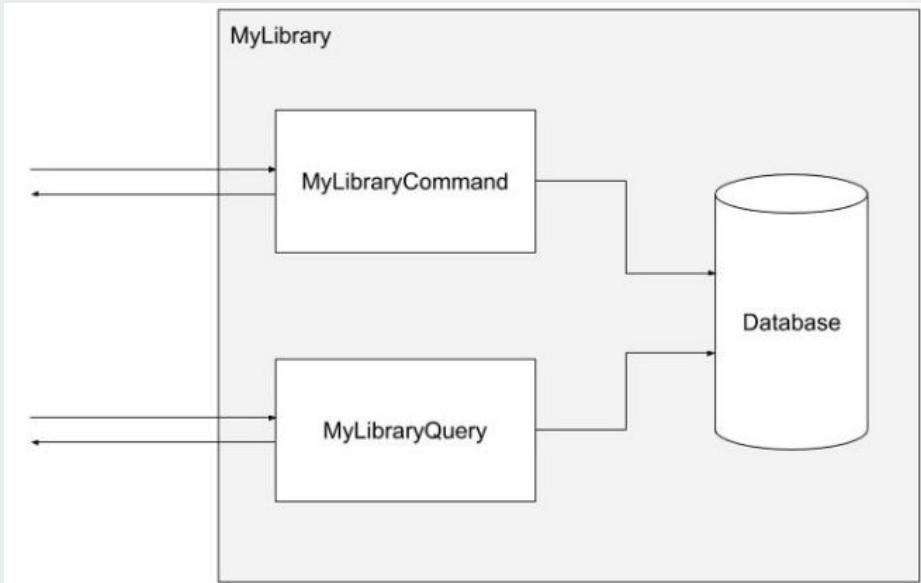
  

MyLibraryQuery
GET getBooks(): List<Book>
GET getBook(id): Book

2 services distincts

## Le pattern architectural CQRS (3 / 3)

- Architecture :



- Les services de lecture et d'écriture peuvent être optimisés de façon indépendante.
- Dans certains cas, deux sources de données peuvent être utilisées avec synchronisation des données.

## Les principes de conception SOLID

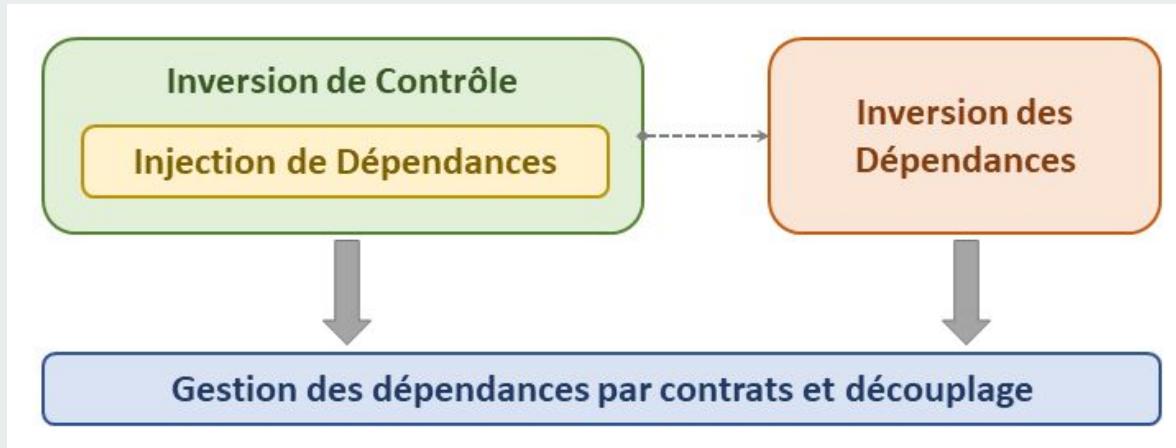
- L'acronyme S.O.L.I.D a été inventé par Michael Feathers à partir des principes de programmation orientée objet identifiés par Robert Cecil Martin en 2002, Ces principes visent à rendre le code plus lisible, facile à maintenir, extensible, réutilisable et sans répétition.
- Les concepts SOLID sont les suivants :
  - Single Responsibility Principle (SRP) : principe de responsabilité unique
  - Open-Closed Principle (OCP) : principe ouvert-fermé
  - Liskov Substitution Principle (LSP) : principe de substitution de Liskov
  - Interface Segregation Principle (ISP) : principe de ségrégation des interfaces
  - Dependency Inversion Principle (DIP) : principe d'inversion des dépendances

## Les principes de conception SOLID

- Exemples avec du code :  
<https://latavernedutesteur.fr/2020/04/28/principes-solid-simplifies-1-5-responsabilite-unique/>
- focus sur le DIP  
<https://www.youtube.com/watch?v=wGqQ1UCIVnI>

## DIP vs IoC vs DI

- DIP : Dependency Inversion Principle, principe d'inversion des dépendances
- IoC : Inversion of Control, inversion de contrôle
- DI : Dependency Injection



Source : <https://www.neosoft.fr/nos-publications/blog-tech/dependances-dip-et-ioc/>

## Vidéo Bonus

- Architecture logicielle : <https://www.youtube.com/watch?v=wtXy77H7QOI>

## Quiz 4b



[https://docs.google.com/forms/d/e/1FAIpQLSfY\\_bB4awMOtj0VMohmsrUcZXn2gJe24-ETpD3z8yOnozJbSA/viewform?usp=dialog](https://docs.google.com/forms/d/e/1FAIpQLSfY_bB4awMOtj0VMohmsrUcZXn2gJe24-ETpD3z8yOnozJbSA/viewform?usp=dialog)