

Proposition d'architecture: Plateforme de réservation d'événements

Table des matières

- [Hypothèses](#)
- [Architecture choisie : Microservices](#)
 - [Description de notre architecture microservices](#)
- [Composants et choix technologiques](#)
 - [Cloud provider : AWS](#)
 - [Backend](#)
 - [Service de réservation : Go + PostgreSQL](#)
 - [Service de paiement : Java + Stripe + PostgreSQL](#)
 - [Service de notification : Novu](#)
 - [Frontend](#)
 - [Application web : landing page + admin dashboard \(Angular\)](#)
 - [Application mobile : Flutter](#)
 - [Authentification & Sécurité](#)
 - [Authentification : Keycloak](#)
 - [Sécurité de l'infrastructure : AWS WAF](#)
 - [Communication & API](#)
 - [Orchestration de Sagas : AWS Step Functions](#)
 - [API Gateway : Java + GraphQL](#)
 - [Message broker: RabbitMQ](#)
 - [Cache des données & Recherche](#)
 - [Système de cache : Redis](#)
 - [Moteur de recherche : Typesense](#)
 - [Data & Observabilité](#)
 - [Dashboard des organisateurs : Metabase](#)
 - [Système de monitoring : AWS + Prometheus/Grafana](#)
 - [DevOps & CI/CD](#)
 - [CI: GitHub Actions](#)
 - [CD: AWS EKS](#)

Hypothèses

La proposition d'architecture logicielle dépend de plusieurs facteurs : le budget, le nombre d'utilisateurs, la taille et la maturité de l'équipe, etc. Dans notre cas d'étude, nous allons faire les suppositions suivantes :

- nous disposons d'un budget conséquent ;
- nous comptons plusieurs milliers d'utilisateurs ;
- nous disposons d'une équipe de 20 à 30 ingénieurs de niveau intermédiaire ou expérimenté.

Architecture choisie : Microservices

En nous basant sur les hypothèses posées (budget conséquent et maturité de l'équipe), nous pouvons nous permettre d'utiliser une **architecture microservices** pour répondre aux contraintes liées au nombre d'utilisateurs.

Description de notre architecture microservices

- Stratégie de décomposition :
 - décomposition par domaine métier (capability-based)
- Stratégie de gestion des données :
 - Base de données par service
 - Pour les requêtes qui span plusieurs services :
 - API composition pour les query
 - Saga orchestration pour les command
 - Style de communication entre microservices :
 - messaging avec des consommateurs idempotent
 - circuit breaker pour augmenter la fiabilité
- Stratégie de déploiement :
 - un service par conteneur combiné à un orchestrateur de conteneur
 - canary releases afin de minimiser les risques

Composants et choix technologiques

Cloud provider : AWS

À mon avis, les géants du cloud (AWS, Azure, GCP) répondent tous à nos besoins. Le choix dépend du type d'infrastructure : on-premise, cloud ou hybride. Pour notre cas d'étude, nous allons utiliser le cloud

avec AWS pour sa flexibilité. Le multi-cloud est une autre option que nous n'allons pas adopter afin d'éviter de tout gérer nous-mêmes, puisque notre budget nous permet d'en déléguer la gestion.

Backend

- Réservation (Go + PostgreSQL)
- Paiement (Java + Stripe + PostgreSQL)
- Notification : Novu

Service de réservation : Go + PostgreSQL

Le service de réservation est un microservice critique pouvant connaître des pics de connexions. Il doit donc être performant et scalable. C'est pourquoi nous utilisons le langage **Go**, reconnu pour sa performance. Une base de données **PostgreSQL** permet d'appliquer des contrôles d'intégrité, renforçant ainsi la cohérence des données.

Service de paiement : Java + Stripe + PostgreSQL

Le service de paiement est un microservice très sensible. Nous avons opté pour **Java** et **PostgreSQL**, des technologies robustes, largement utilisées et faciles à intégrer avec des solutions existantes. **Stripe** est une infrastructure financière très flexible, facile à prendre en main, et bien adaptée à notre modèle économique.

Service de notification : Novu

La notification est un point essentiel pour cette plateforme. C'est pourquoi nous avons opté pour une infrastructure qui regroupe tous les canaux : in-app, email, chat, push notification, SMS, etc. **Novu** est un choix pertinent, car il est open source et permet de définir des workflows assimilables à des pipelines CI/CD pour la notification, rendant les alertes pertinentes, personnalisées et ciblées.

Frontend

- Web (Angular SSR)
- Mobile (Flutter)

Application web : landing page + admin dashboard (Angular)

Pour la landing page, il faut optimiser les métriques **Core Web Vitals**. Nous allons donc utiliser une technologie intégrant le SSR (Server Side Rendering), le lazy loading, le caching, etc., ce qui est

disponible avec **Angular**. L'utilisation de CDN permet également d'améliorer les performances.

L'interface des organisateurs sera une application web pouvant croître en complexité (dashboard, gestion de comptes, notifications, comptabilité, etc.). Il nous faut une technologie avec un écosystème complet, une structure rigoureuse, et un state management robuste. **Angular** est un bon choix car il répond à tous ces critères, contrairement à des alternatives comme **React**, qui n'offrent pas de structure rigide et laissent ces choix au développeur.

Application mobile : Flutter

L'application mobile permettra de réserver des événements et sera destinée au grand public. L'idéal est qu'elle fonctionne sur un maximum de plateformes (iOS, Android, HarmonyOS, etc.). C'est pourquoi nous avons choisi **Flutter**.

Authentification & Sécurité

- Authentification: Keycloak
- Sécurité de l'infrastructure: AWS WAF

Authentification : Keycloak

Puisqu'il s'agit d'une plateforme de réservation d'événements, il est important de faciliter l'accès à nos services tout en maintenant un bon niveau de sécurité. Il nous faut une technologie qui implémente **OAuth2.0**, pour permettre à nos utilisateurs de créer un compte ou de se connecter via leurs comptes existants sur d'autres plateformes. **Keycloak** est un choix incontournable grâce à sa robustesse et sa grande communauté, surpassant des alternatives comme **AWS Cognito**.

Sécurité de l'infrastructure : AWS WAF

AWS met à disposition plusieurs services de sécurité à différents niveaux (application, réseau, accès, etc.). Parmi les outils essentiels, on peut citer **AWS WAF**, et **AWS Secrets Manager**.

Communication & API

- API Gateway (GraphQL + Java)
- Sagas (AWS Step Functions)
- Message broker (RabbitMQ)

Orchestration de Sagas : AWS Step Functions

Nous allons utiliser **AWS Step Functions** pour orchestrer les transactions saga. Nous évitons les moteurs de workflow comme **Camunda** en raison de leur lourdeur. Mettre en place un microservice orchestrateur est une autre option, mais elle peut s'avérer très complexe et sujette à erreurs.

API Gateway : Java + GraphQL

Le front communiquera avec l'API Gateway en utilisant du **GraphQL**, qui permet d'éviter l'over-fetching, de bénéficier d'un schéma typé, et de souscrire aux mises à jour de données en temps réel.

Message broker: RabbitMQ

Pour la file d'attente de messages nous allons utiliser **RabbitMQ** qui a une architecture simple et mieux adapté à nos besoins. **Kafka** est une alternative intéressante cependant il est très complexe et offre un système de messagerie durable et d'autres fonctionnalités avancées qu'on ne va pas utiliser vu que nos sagas utilisent déjà un orchestrateur et on a déjà un gateway pour les jointures inter-services.

Cache des données & Recherche

- Système de cache: Redis
- Moteur de recherche: Typesense

Système de cache : Redis

Pour les services métier, il est nécessaire de mettre en place un système de cache. Notre choix se porte sur **Redis**, qui est flexible, performant et dispose d'une grande communauté.

Moteur de recherche : Typesense

Un moteur de recherche améliore l'expérience utilisateur grâce à l'auto-complétion, la recherche poussée (Geo-search pour les événements à proximité, Semantic Search, etc.). Notre choix se porte sur **Typesense** car il est open source, plus léger que la plupart des moteurs de recherche comme Elasticsearch, et propose des fonctionnalités intéressantes comme la mémoire à long terme pour les LLMs et la visualisation des données (graphiques, tableaux).

Data & Observabilité

- Dashboard des organisateurs: Metabase
- Système de monitoring: CloudWatch + Prometheus/Grafana

Dashboard des organisateurs : Metabase

Pour l'interface des organisateurs, il est pertinent de mettre en place des dashboards pour la prise de décision. **Metabase** est un bon outil open source pour générer des dashboards. Il est très léger et peut se connecter à plusieurs sources de données.

Système de monitoring : AWS + Prometheus/Grafana

Pour le monitoring, nous allons utiliser **CloudWatch**, proposé par AWS. L'objectif est de déléguer une partie du monitoring au cloud provider. **CloudWatch** offre de nombreux avantages comme l'automatisation du scaling via des seuils, le système d'alerte, etc. **Prometheus** et **Grafana** seront également utilisés pour pallier les limites de CloudWatch.

DevOps & CI/CD

- CI: GitHub Actions
- CD: AWS EKS

CI: GitHub Actions

Pour le CI, nous allons utiliser **GitHub Actions**, très puissant et bénéficiant d'une large communauté. Sa flexibilité, comme la possibilité de réagir à la création d'une issue, n'est pas toujours retrouvée dans d'autres outils d'intégration.

CD: AWS EKS

Pour le CD, nous utiliserons **AWS EKS**, très compatible avec nos microservices conteneurisés, ce qui décharge l'équipe de la maintenance du cluster.