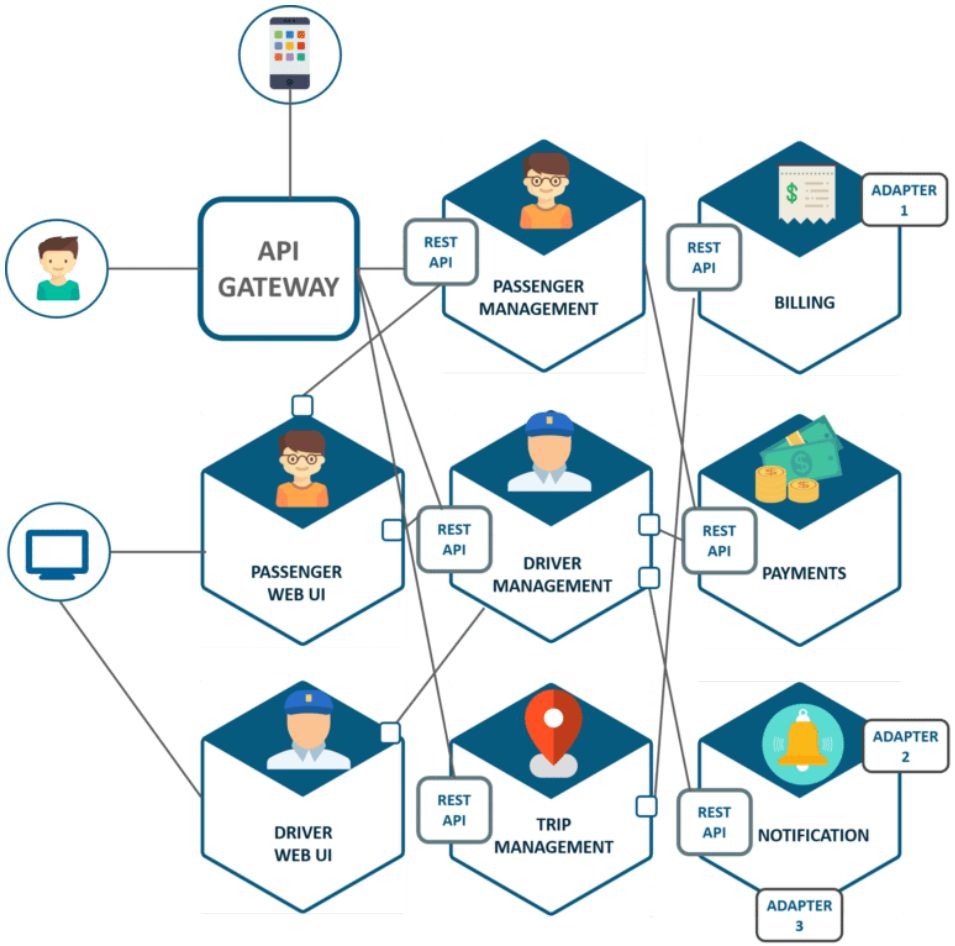
ALOM

PATTERNS ORIENTÉS CLOUD

UBER


API
GATEWAY
REST
API
PASSENGER
MANAGEMENT
REST
API
BILLING
ADAPTER
1
PASSENGER
WEB UI
REST
API
DRIVER
MANAGEMENT
REST
API
PAYMENTS
DRIVER
WEB UI
REST
API
TRIP
MANAGEMENT
REST
API
NOTIFICATION
ADAPTER
2
ADAPTER
3

PROBLÉMATIQUES :

Comment déployer les services ?

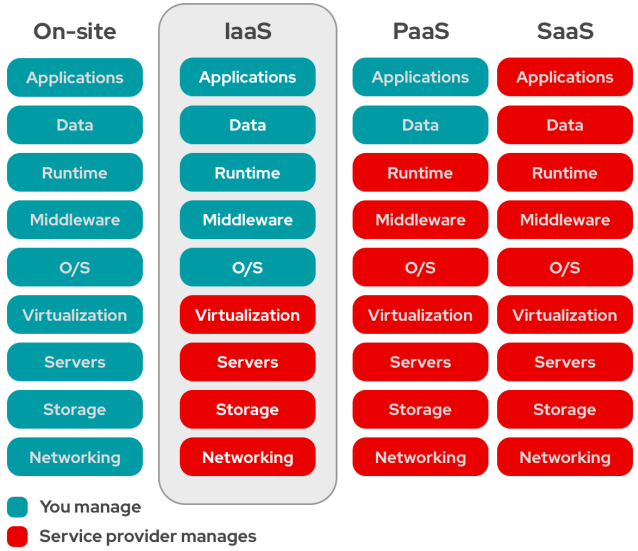
Comment gérer la configuration des services ?

Comment analyser l'enchaînement des appels ?

Comment connaître l'état de l'application ?

CLOUD "AS A SERVICE"

- IaaS : Infrastructure as a Service
- PaaS : Platform as a Service
- SaaS : Software as a Service



CLOUD 🇫🇷

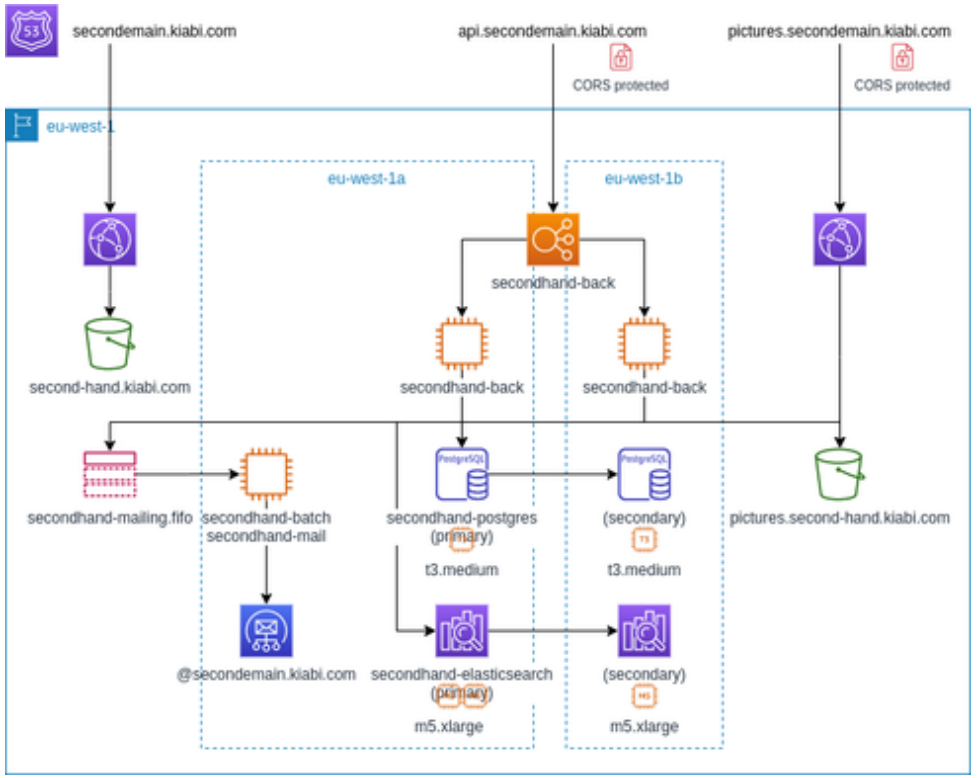
- IaaS : VM, Disks, Load-Balancers
  - OVH Cloud
  - Scaleway
  - Outscale
- PaaS : Database, Middleware, Object Storage, Runtime Java / Container, Functions
  - OVH : Databases & k8s, Object Storage
  - Scaleway: Databases & k8s, Object Storage, Serverless Functions
  - Clever-Cloud: PaaS, Databases, Object Storage

CLOUD 🇺🇸

- IaaS : VM, Disks, Load-Balancers
  - AWS EC2
  - GCP GCE
  - Azure VM
- PaaS : Database, Middleware, Object Storage, Runtime Java / Container, Functions
  - AWS : RDS Databases, EKS, S3 Object Storage, AWS Lambdas
  - GCP Cloud SQL, GKE, GCS Object Storage, Cloud Functions, App Engine, Pub/Sub
  - Azure SQL Database, AKS, Azure Blob

UNE ARCHITECTURE CLOUD

Une application Java dans un assemblage de services Cloud



UNE ARCHITECTURE CLOUD

- DNS (Route53)
- Load Balancers (ALB)
- VM (EC2) ⬅️ notre code est ici
- Base de données managée (PostgreSQL)
- Cache managé (ElasticSearch)
- CDN (CloudFront)
- Stockage objet (S3)
- Messaging (SQS)
- Mailing (SES)

PATTERNS D'ARCHITECTURE

CONFIGURATION EXTERNALISÉE

CONFIGURATION EXTERNALISÉE

Permet d'exécuter un service dans multiples environnements sans modifications

- Accès BDD
- Gestion sécurité
- etc...

Plusieurs stratégies

- Profiles : un fichier de configuration par profil
- Variables d'environnement
- Serveur de configuration

CONFIGURATION EXTERNALISÉE EN SPRING

Profils

- Activation par la properties `spring.profiles.active`
- ou variable d'environnement `SPRING_PROFILES_ACTIVE`

Permet de charger un fichier `application-{profile}.properties` en plus du `application.properties`.

Exemples: `application-local.properties` et `application-prod.properties` contenant des

CONFIGURATION EXTERNALISÉE CHEZ SPRING

Variables d'environnement

Toutes les properties Spring peuvent être surchargées par des variables d'environnement

- Les `.` sont remplacés par des `_`.
- Tout est mis en majuscule
- Le camel-case est convertit en `_`

CONFIGURATION EXTERNALISÉE CHEZ SPRING

Variables d'environnement

Exemples:

- `server.port=8080` ➡️ `SERVER_PORT=8080`
- `trainer.service.username=vegeta` ➡️ `TRAINER_SERVICE_USERNAME=vegeta`
- `trainer.serviceUrl=https://someurl:8080` ➡️ `TRAINER_SERVICE_URL=https://someurl:8080`

CONFIGURATION EXTERNALISÉE CHEZ SPRING

Interpolation de properties

Il est possible dans des properties d'en utiliser d'autres

```
trainer.service.host=someHost
trainer.service.port=8080
trainer.service.url=https://${trainer.service.host}:${trainer.
```

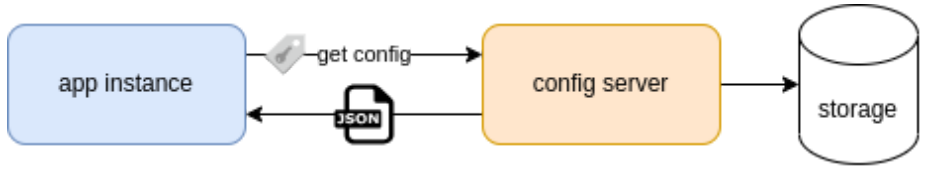
SERVEURS DE CONFIGURATION

Chargement de properties gérées dans un serveur

Spring requête le serveur au démarrage pour charger les properties

- Vault / Consul
- Azure Key Vault, AWS/GCP Secret Manager
- Kubernetes ConfigMaps / Secrets

SERVEURS DE CONFIGURATION



Intérêts :

- partage de configuration entre plusieurs apps
- rotation de mots de passes sans devoir rebuilder l'appli
- centralisation et contrôle d'accès

PATTERNS D'ARCHITECTURE

CENTRALISATION DES LOGS ET CORRELATION

CENTRALISATION DES LOGS

Dans un environnement load-balancé, les requêtes d'un utilisateur peuvent être traitées par n'importe quel serveur.

Dans un environnement cloud, on ne peut pas forcément accéder aux machines pour consulter les fichiers de log.

Dans un environnement conteneurisé, on ne peut pas forcément accéder aux logs des containers (kubernetes...)

CENTRALISATION DES LOGS

On envoie tous les logs dans un service dédié

- un service lit les fichiers de log ou la sortie standard `stdout` et envoie les lignes au serveur
- les logs sont indexés et conservés
- une IHM permet de les consulter

CENTRALISATION DES LOGS

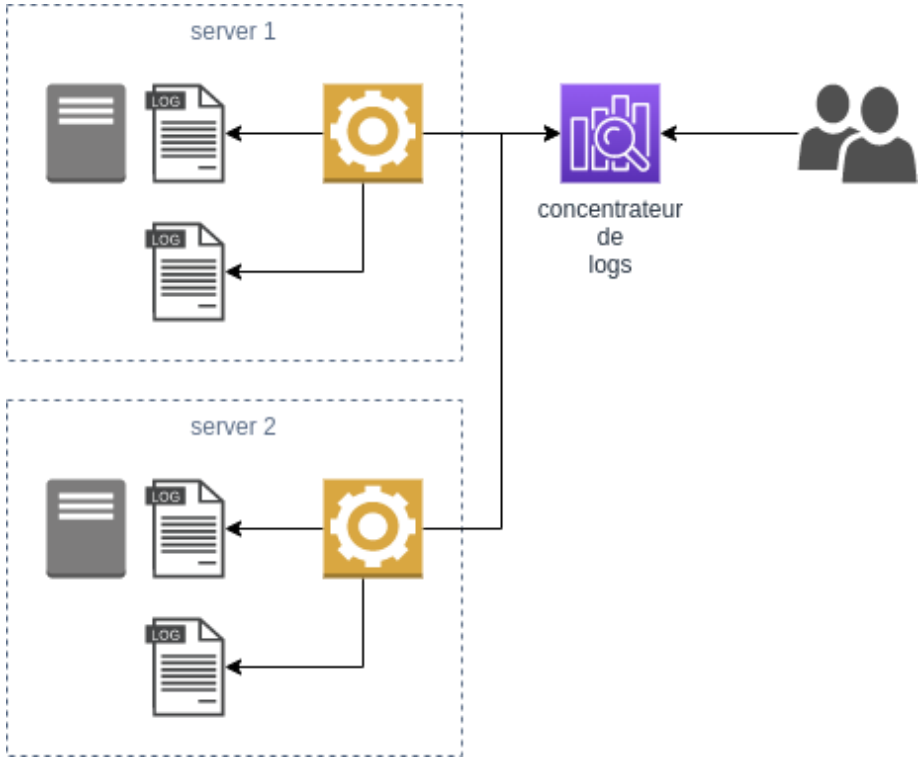
LOGS JSON

Pour rendre les logs _requêtables_, ils sont souvent parsés :

```
2023-11-28T14:05:57.429+01:00  INFO 62385 --- [main] TrainerAp
```

On configure parfois les loggers pour émettre du JSON directement utilisable en centralisation.

CENTRALISATION DES LOGS



CENTRALISATION DES LOGS

Produit connus :

Stack "ELK"

- Elasticsearch : indexation des logs, et recherche "full-text"
- Logstash : Parsing des fichiers de logs, et envoi à Elasticsearch
- Kibana : IHM de consultation d'Elasticsearch : recherche, dashboards...

CORRELATION DES LOGS

Observer la séquencialité des appels

Observer les logs d'un même utilisateur

Trouver des points de contention

Aide au debugging

CORRELATION DES LOGS

Correlation des appels via des Headers HTTP

Création d'un id pour chaque requête reçue

Transmission de l'id à chaque requête envoyée

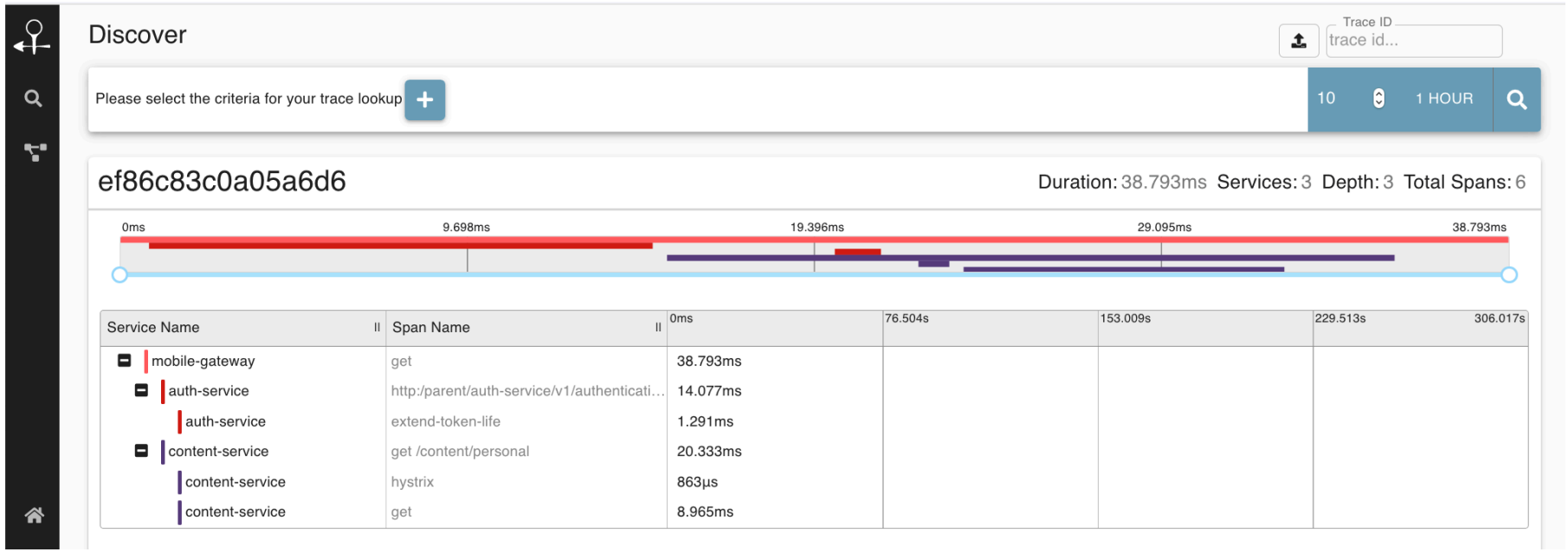
Envoi des traces à un outil centralisé

CORRELATION DES LOGS

Spring Cloud Sleuth permet de gérer ces correllation (il modifie les RestTemplate pour transmettre ces fameux headers).

Zipkin permet de collecter/consulter ce type d'information

CORRELATION DES LOGS



PATTERNS D'ARCHITECTURE

OBSERVABILITÉ / MÉTRIQUES

MÉTRIQUES

Observer la santé des services

- healthcheck : est-ce que le service répond, est-ce que la BDD est bien connectée
- trace : récupérer les dernières requêtes HTTP traitées
- metrics : consommations mémoire / CPU

Métriques métier (~analytics)

- Combien d'inscriptions au site
- Combien de commandes passées
- ...

MÉTRIQUES

Exposition des métriques dans une application spring-boot

Utilisation de spring-boot-actuator

Expose des métriques basiques de nos applications/api

MÉTRIQUES

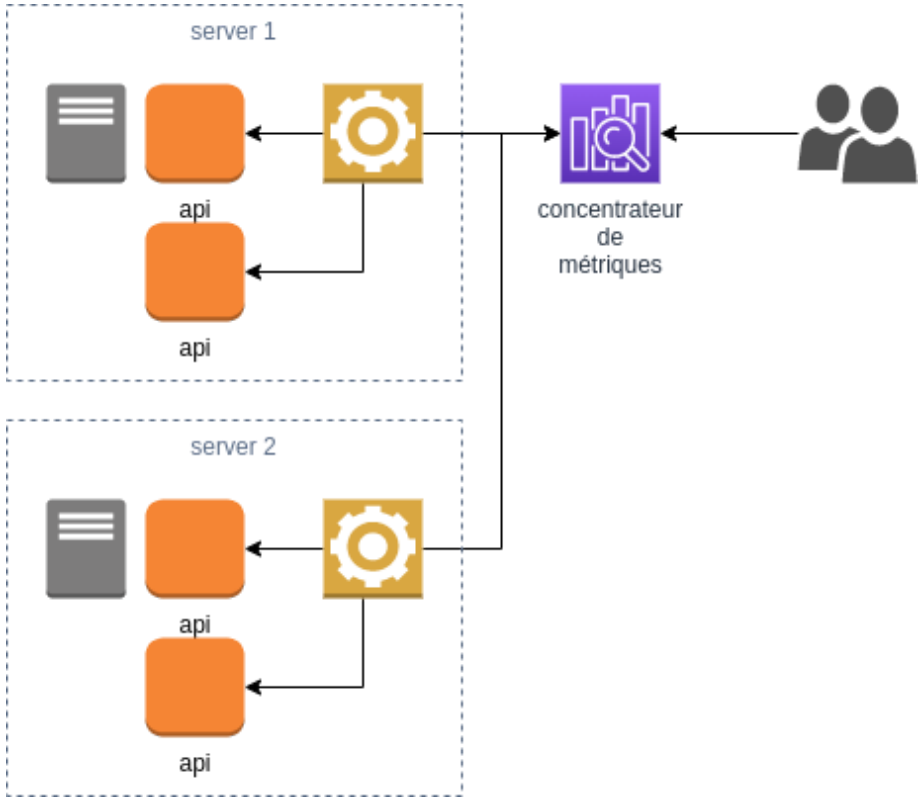
COLLECTE DES MÉTRIQUES ET EXPLOITATION

Comme pour les logs, les métriques peuvent être envoyées à un serveur dédié pour être consultées

Centralisation des métriques

MÉTRIQUES

Même principe que pour les logs



MÉTRIQUES

Produits connus :

Stack "Prometheus/Grafana"

- Prometheus : Concentration des métriques (BDD time/series)
- Grafana : Affichage sous forme de graphes, alerting

MÉTRIQUES

Agir en fonction des métriques

Pris en charge par les orchestrateurs de containers (kubernetes par exemple)

- healthcheck KO => redémarrer le service
- consommation mémoire / CPU élevée => déployer une instance supplémentaire du service (scale up)

OPENTELEMETRY

- Standardiser la collecte des signaux d'observabilité : **traces**, **métriques**, **logs**
- Corréler bout-en-bout les appels via la propagation de contexte (W3C Trace Context : `traceId`, `spanId`)
- Découpler le code applicatif des outils de stockage/visualisation (vendor-neutral)
- Offrir des SDKs et de l’auto-instrumentation pour de nombreux langages (Java, JS, Python, …)
- Utiliser un protocole ouvert et efficace pour l’export : **OTLP** (gRPC/HTTP)

STACK OTEL STANDARD

Architecture type de collecte et d'exploitation

```
Application (SDK/Agent OTEL)
   └─ instrumentation auto/manuelle → OTLP
        └─ OTEL Collector (ingest/process/export)
             ├─ Traces    → Jaeger / Tempo / Zipkin
             ├─ Métriques→ Prometheus / Mimir / Cloud vendor
             └─ Logs      → Loki / Elasticsearch

Visualisation : Grafana (traces/metrics/logs), Kibana (logs)
```

- **OTEL SDK / Agents** : ajout de spans, attributs, métriques; propagation des headers
- **OTEL Collector** : composant central pour recevoir, transformer, exporter (pipeline)
- **Backends** : stockage/consultation selon le type de

TP

Patterns cloud