


# Conception d'applications conteneurisées pour OpenShift

Principes de conception, microservices et Dockerfile avancés

 Principes KISS, DRY, YAGNI, SoC

 Microservices

 Dockerfile avancés



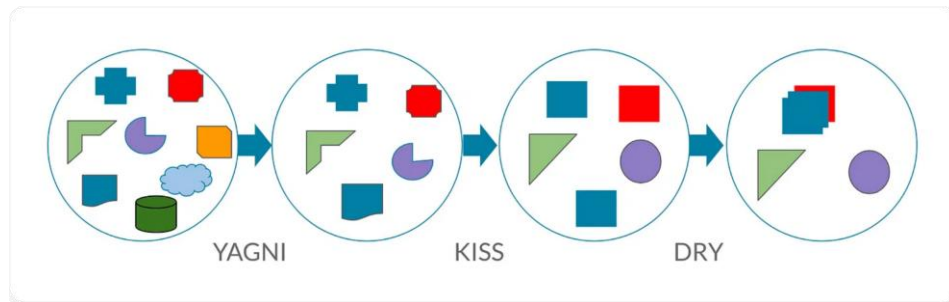
**OPENS**HIFT

# Introduction aux principes de conception

## Pourquoi des principes de conception ?

Les principes de conception logicielle sont des lignes directrices qui aident les développeurs à créer des applications **maintenables**, **évolutives** et **robustes**

Dans le contexte des applications conteneurisées pour OpenShift, ces principes sont encore plus importants pour garantir une architecture adaptée aux environnements cloud-native.



- ✓ **KISS** - Keep It Simple, Stupid
- ✓ **DRY** - Don't Repeat Yourself
- ✓ **YAGNI** - You Aren't Gonna Need It
- ✓ **SoC** - Separation of Concerns

# Principe KISS (Keep It Simple, Stupid)

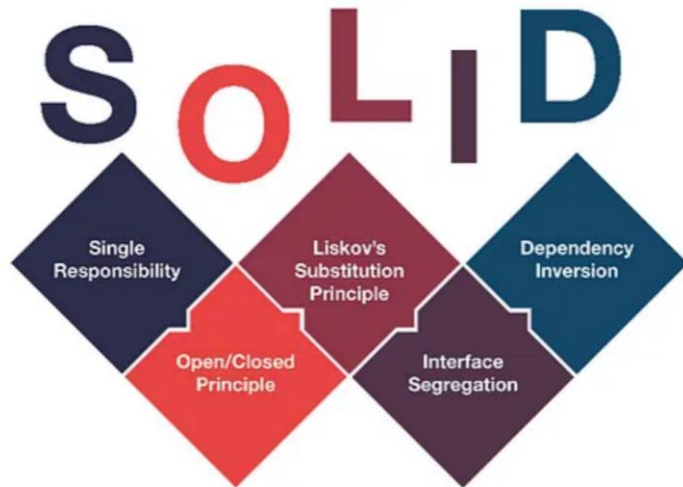
## La simplicité avant tout

Le principe KISS encourage à concevoir des solutions simples et directes, en évitant la complexité inutile. La simplicité rend le code plus facile à comprendre, à maintenir et à déboguer.

- ✓ Facilite la compréhension du code par tous les membres de l'équipe
- ✓ Réduit les risques d'erreurs et de bugs
- ✓ Simplifie la maintenance et les évolutions futures
- ✓ Améliore les performances en éliminant le code superflu

### Application dans OpenShift :

- Créer des images de conteneur légères et spécialisées
- Limiter le nombre de dépendances
- Concevoir des microservices avec une responsabilité unique
- Utiliser des configurations simples et explicites



### Exemple de Dockerfile suivant KISS :

```
FROM node:14-alpine
WORKDIR /app
COPY package*.json ./
RUN npm ci --only=production
COPY . .
EXPOSE 3000
CMD ["node", "app.js"]
```

# Principe DRY (Don't Repeat Yourself)

## Éviter la duplication de code

Le principe DRY stipule que

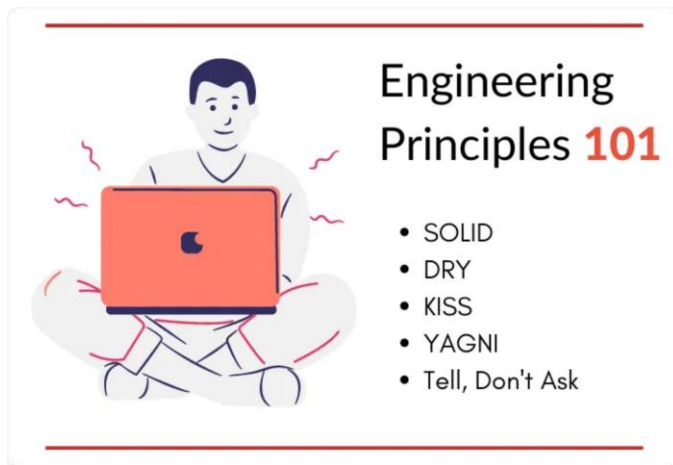
**"chaque élément de connaissance doit avoir une représentation unique, non ambiguë et faisant autorité dans un système"**

En pratique, cela signifie éviter la duplication de code et de logique en créant des abstractions réutilisables.

- ✓ Réduit la taille du code et facilite sa maintenance
- ✓ Limite les risques d'incohérence lors des modifications
- ✓ Améliore la lisibilité et la compréhension du code
- ✓ Favorise la création de composants réutilisables

### Application dans OpenShift :

- Utiliser des images de base communes
- Créer des bibliothèques partagées entre microservices
- Définir des templates et des opérateurs réutilisables
- Centraliser les configurations avec ConfigMaps



#### Non-DRY

```
# Service 1
FROM node:14
WORKDIR /app
RUN apt-get update && \
    apt-get install -y curl
COPY ..

# Service 2
FROM node:14
WORKDIR /app
RUN apt-get update && \
    apt-get install -y curl
COPY ..
```

#### DRY

```
# Image de base commune
FROM node:14
WORKDIR /app
RUN apt-get update && \
    apt-get install -y curl

# Services
FROM base-image
WORKDIR /app
COPY ..
```

# Principe YAGNI (You Aren't Gonna Need It)

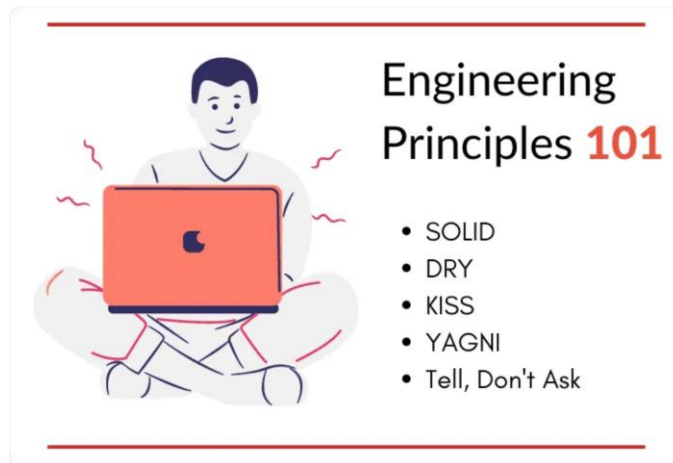
## Développer uniquement ce qui est nécessaire

Le principe YAGNI recommande de ne pas ajouter de fonctionnalités tant qu'elles ne sont pas strictement nécessaires. Il s'agit d'éviter le sur-développement et de se concentrer sur les besoins réels et immédiats.

- ✓ Réduit la complexité du code et de l'architecture
- ✓ Économise du temps et des ressources de développement
- ✓ Minimise la dette technique liée aux fonctionnalités inutilisées
- ✓ Permet de s'adapter plus facilement aux changements d'exigences

### Application dans OpenShift :

- Créer des microservices avec des fonctionnalités essentielles
- Éviter d'implémenter des fonctionnalités "au cas où"
- Commencer avec des configurations simples et les enrichir selon les besoins
- Utiliser une approche itérative pour le développement



Approche	Résultat
✓ "On pourrait avoir besoin de X plus tard"	Complexité accrue, maintenance difficile
✗ "Implémentons X quand nous en aurons besoin"	Solution adaptée aux besoins réels
✗ Anticiper tous les cas d'utilisation possibles	Surcharge de développement, fonctionnalités inutilisées
✓ Développer pour les besoins actuels	Livraison plus rapide, code plus simple

# Principe SoC (Separation of Concerns)

## Séparer les responsabilités

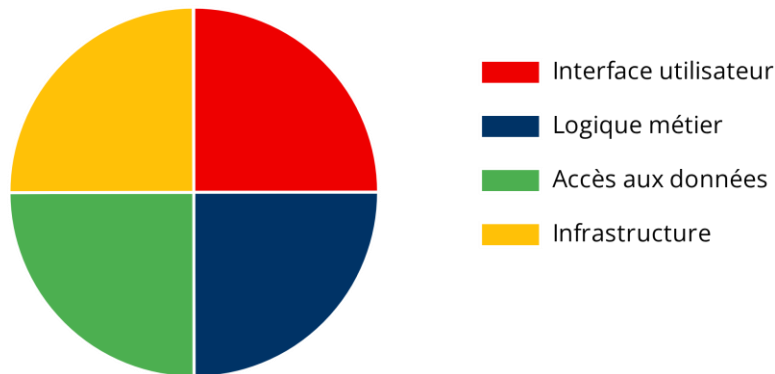
Le principe SoC consiste à diviser un programme en sections distinctes, chacune traitant un aspect spécifique. Chaque composant doit avoir une responsabilité unique et bien définie.

- ✓ Facilite la maintenance en isolant les changements
- ✓ Permet le développement parallèle par différentes équipes
- ✓ Améliore la testabilité des composants individuels
- ✓ Favorise la réutilisation des composants

### Application dans OpenShift :

- Séparer l'application en microservices distincts
- Isoler la logique métier de l'infrastructure
- Utiliser des ConfigMaps pour externaliser la configuration
- Séparer les environnements avec des namespaces

### Séparation des préoccupations



### Exemple d'architecture SoC sur OpenShift :






- Frontend : Interface utilisateur (React, Angular)
- API Gateway : Routage et authentification
- Services métier : Logique applicative
- Services de données : Accès aux bases de données
- Services d'infrastructure : Logging, monitoring

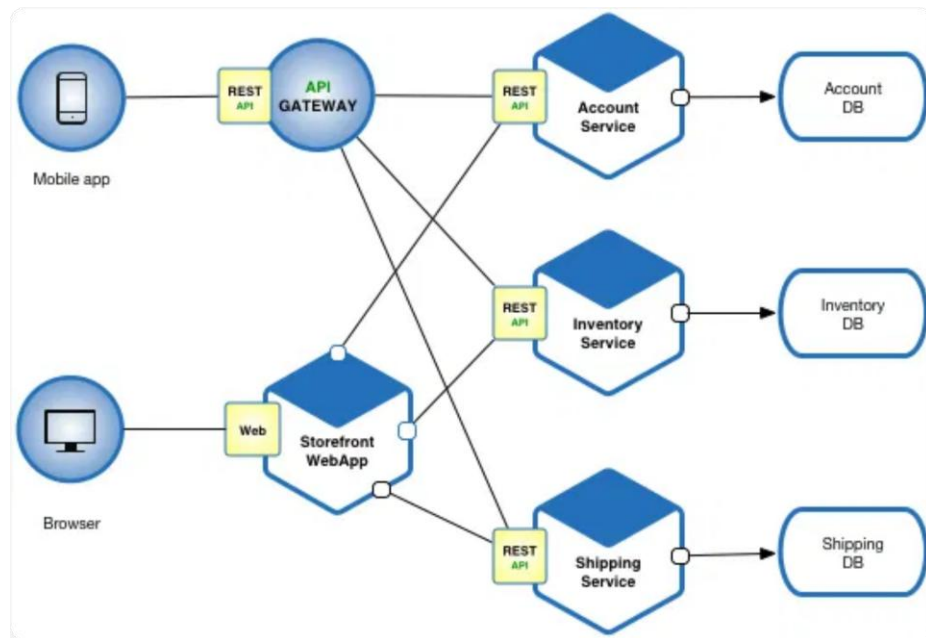
# Introduction aux microservices

## Qu'est-ce qu'une architecture microservices ?

Une architecture microservices est une approche de développement logiciel qui structure une application comme un ensemble de **services faiblement couplés** chacun implémentant une fonctionnalité métier spécifique.

Contrairement aux applications monolithiques, les microservices permettent un développement, un déploiement et une mise à l'échelle indépendants, ce qui les rend particulièrement adaptés aux environnements cloud comme OpenShift.






-  Services autonomes avec responsabilité unique
-  Communication via API bien définies
-  Base de données dédiée par service
-  Déploiement et mise à l'échelle indépendants
-  Développement par équipes autonomes

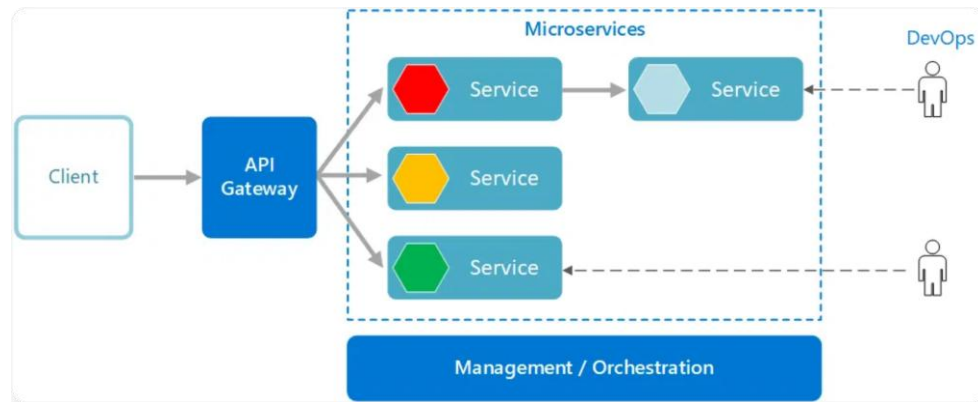


# Architecture des microservices

## Composants d'une architecture microservices

L'architecture microservices décompose une application en services indépendants, chacun ayant une responsabilité unique et communiquant via des API bien définies.

-  **Services autonomes** : Chaque service est développé, déployé et mis à l'échelle indépendamment
-  **API Gateway** : Point d'entrée unique qui route les requêtes vers les services appropriés
-  **Base de données par service** : Chaque service gère ses propres données
-  **Service Discovery** : Mécanisme permettant aux services de se localiser dynamiquement
-  **Circuit Breaker** : Protège le système contre les défaillances en cascade





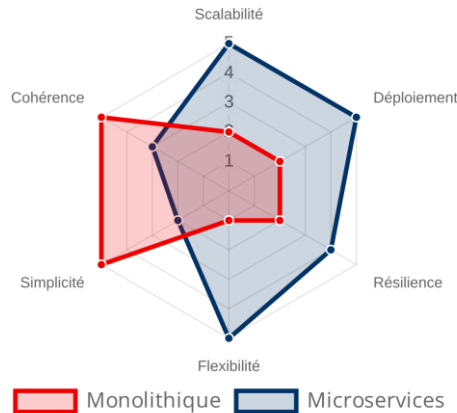
# Avantages et défis des microservices

## Pourquoi choisir les microservices ?

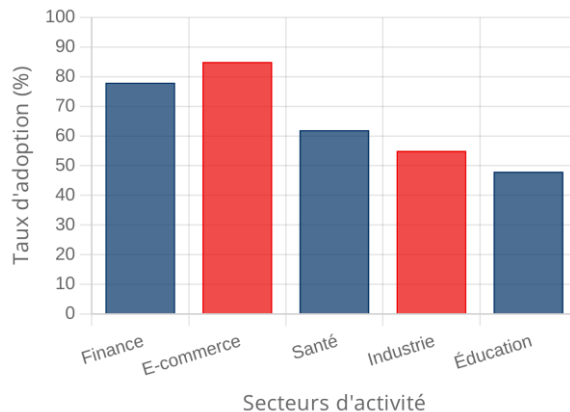
L'architecture microservices offre de nombreux avantages, mais présente également des défis qu'il est important de comprendre avant de l'adopter.

- ✓ **Scalabilité indépendante** - Mise à l'échelle précise des services
- ✓ **Déploiement continu** - Livraison plus rapide et moins risquée
- ✓ **Résilience** - Isolation des défaillances
- ✓ **Flexibilité technologique** - Choix de la meilleure technologie
- ⚠ **Complexité distribuée** - Gestion des communications
- ⚠ **Cohérence des données** - Maintien de la cohérence
- ⚠ **Monitoring et débogage** - Traçage des requêtes

## Comparaison : Monolithique vs Microservices



## Adoption des microservices par secteur



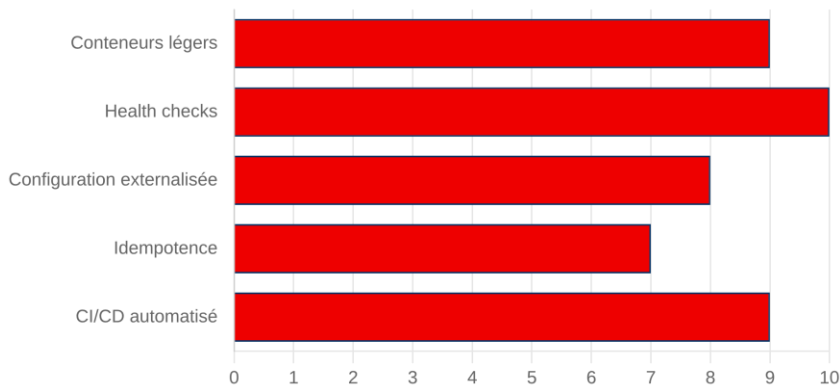
# Bonnes pratiques pour les microservices sur OpenShift

## Optimiser vos microservices pour OpenShift

OpenShift offre un environnement idéal pour déployer des microservices, mais pour en tirer pleinement parti, il est important de suivre certaines bonnes pratiques.

- ✓ **Conteneurs légers** - Utilisez des images de base minimales et optimisez les couches Docker
- ✓ **Health checks** - Implémentez des endpoints de liveness et readiness
- ✓ **Configuration externalisée** - Utilisez les ConfigMaps et Secrets
- ✓ **Idempotence** - Concevez vos services pour qu'ils puissent être redémarrés sans effets secondaires

Importance des bonnes pratiques pour les microservices



### Exemple de Health Check dans OpenShift :

```
apiVersion: apps/v1
kind: Deployment
spec:
  template:
    spec:
      containers:

      livenessProbe:
        httpGet:
          path: /health
          port: 8080
```

# Introduction aux travaux pratiques

## Création d'images de conteneur avancées

Dans cette partie pratique, nous allons explorer la création d'images de conteneur optimisées pour OpenShift en utilisant des instructions avancées de Dockerfile.

Ces travaux pratiques vous permettront d'appliquer les principes de conception que nous avons étudiés (KISS, DRY, YAGNI, SoC) dans le contexte de la conteneurisation d'applications.

- 🎯 Maîtriser les instructions avancées de Dockerfile
- 🎯 Optimiser les images pour la sécurité et les performances
- 🎯 Appliquer les bonnes pratiques pour OpenShift
- 🎯 Créer des images multi-étapes pour différents environnements

## Best Practices and Tips for Writing a Dockerfile



### 1 Analyse d'un Dockerfile existant

Identification des problèmes et opportunités d'amélioration

### 2 Optimisation avec multi-stage builds

Séparation des étapes de build et de production

### 3 Implémentation des instructions avancées

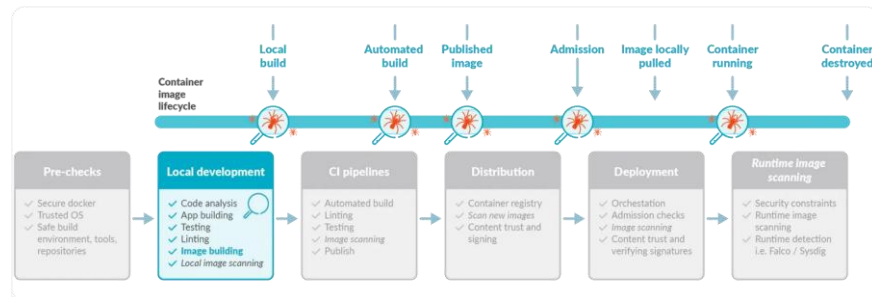
Utilisation de HEALTHCHECK, ARG, ONBUILD, etc.

# Instructions avancées de Dockerfile

## Au-delà des instructions de base

Les Dockerfile avancés utilisent des instructions spécifiques pour optimiser la taille des images, améliorer la sécurité et faciliter la maintenance. Ces techniques sont particulièrement importantes pour les applications déployées sur OpenShift.

- 📦 **Multi-stage builds** : Utilise plusieurs étapes pour réduire la taille finale de l'image
- ↔️ **ARG et ENV** : Paramétrise les builds et définit des variables d'environnement
- 💓 **HEALTHCHECK** : Vérifie l'état de santé du conteneur en cours d'exécution
- 👤 **USER** : Définit l'utilisateur non-root pour améliorer la sécurité
- 🔊 **VOLUME** : Définit les points de montage pour la persistance des données



### # Exemple de multi-stage build

```
FROM node:14 AS build
WORKDIR /app
COPY package*.json ./
RUN npm install
COPY . .
RUN npm run build


FROM nginx:alpine
COPY --from=build /app/dist /usr/share/nginx/html
HEALTHCHECK --interval=30s --timeout=3s \
  CMD wget -q -O- http://localhost/ || exit 1

USER nginx
EXPOSE 80
CMD ["nginx", "-g", "daemon off;"]
```

# Exemple de Dockerfile avancé

## Application Node.js avec multi-stage build

Voici un exemple de Dockerfile avancé pour une application Node.js, utilisant plusieurs techniques modernes pour créer une image optimisée pour OpenShift.

 **Multi-stage build** : Sépare les étapes de build et de production

 **Utilisateur non-root** : Améliore la sécurité

 **HEALTHCHECK** : Vérification de l'état de santé

 **ARG et ENV** : Paramétrage flexible de l'image

```
WORKDIR /app

# Copier les fichiers package.json et package-lock.json
COPY package*.json ./

# Installer les dépendances
RUN npm install

# Copier le reste des fichiers de l'application
COPY . .

# Construire l'application
RUN npm run build

# Étape 2: Environnement de production
FROM node:18-alpine

# Définir le répertoire de travail
WORKDIR /app

# Copier les fichiers package.json et package-lock.json
COPY package*.json ./

# Installer uniquement les dépendances de production
RUN npm install --only=production

# Copier les fichiers de build de l'application depuis l'étape de build
COPY --from=builder /app/dist ./dist
COPY --from=builder /app/node_modules ./node_modules

# Exposer le port sur lequel l'application va tourner
EXPOSE 3000

# Commande pour démarrer l'application
CMD ["node", "dist/main.js"]
```

# Bonnes pratiques pour les Dockerfile

## Optimiser vos images pour OpenShift

Suivre les bonnes pratiques pour la création de Dockerfile permet d'obtenir des images plus légères, plus sécurisées et mieux adaptées à l'environnement OpenShift.

- ✓ **Minimiser les couches** : Combiner les commandes RUN pour réduire le nombre de couches
- ✓ **Utiliser .dockerignore** : Exclure les fichiers inutiles du contexte de build
- ✓ **Privilégier les images officielles** : Utiliser des images de base maintenues et sécurisées
- ✓ **Utiliser des tags spécifiques** : Éviter les tags flottants comme 'latest'
- ✓ **Nettoyer après installation** : Supprimer les caches et fichiers temporaires

### Exemple d'optimisation :

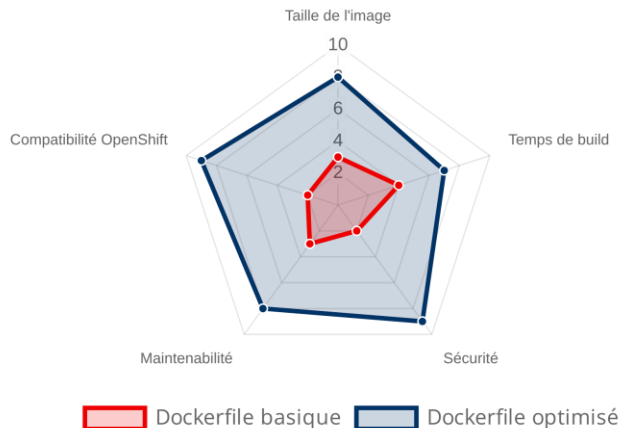
#### # À éviter

```
RUN apt-get update
RUN apt-get install -y curl
RUN apt-get clean
```

#### # Recommandé

```
RUN apt-get update && apt-get install -y curl && \
apt-get clean && rm -rf /var/lib/apt/lists/*
```

## Impact des bonnes pratiques Dockerfile



## À faire vs. À éviter

- ✓ Utiliser des images de base légères (alpine)
- ✗ Utiliser des images complètes avec des outils inutiles
- ✓ Exécuter en tant qu'utilisateur non-root
- ✗ Exécuter en tant que root (problèmes de sécurité)
- ✓ Définir des HEALTHCHECK
- ✗ Ignorer la surveillance de l'état du conteneur
- ✓ Optimiser l'ordre des instructions (cache)
- ✗ Modifier fréquemment les fichiers qui invalident le cache

# Résumé et conclusion

## Points clés à retenir

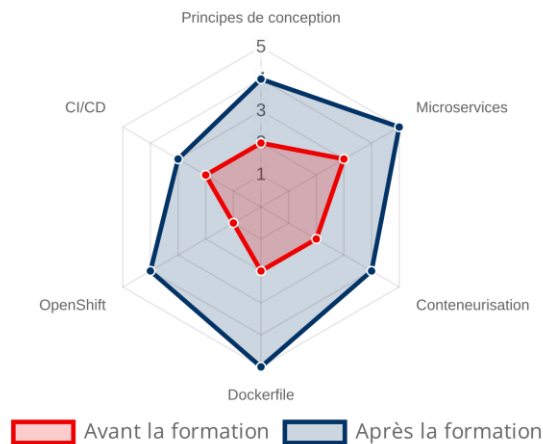
La conception d'applications conteneurisées pour OpenShift nécessite une approche méthodique qui combine les principes de conception logicielle et les bonnes pratiques de conteneurisation.

- ✓ **Principes KISS, DRY, YAGNI et SoC** : Fondamentaux pour créer des applications maintenables et évolutives
- ✓ **Architecture microservices** : Approche idéale pour les applications cloud-native sur OpenShift
- ✓ **Dockerfile avancés** : Techniques essentielles pour optimiser les images de conteneur
- ✓ **Bonnes pratiques OpenShift** : Adaptations spécifiques pour tirer pleinement parti de la plateforme

### Prochaines étapes

- Appliquer ces principes à vos projets existants
- Explorer les modèles OpenShift (Templates)
- Approfondir les stratégies de déploiement avancées

## Progression des compétences



## Questions ?

Merci pour votre attention !