🎓 Formation ArgoCD — Déploiement continu GitOps

Support de cours complet

1. Module 0 : Accueil et préparation — 30 min
2. Module 1 : GitOps et architecture d'ArgoCD — 1 h
3. Module 2 : Installation d'ArgoCD sur Kubernetes — 1 h 45
4. Module 3 : Configuration et sécurité des accès — 3 h
5. Module 4 : Manifestes d'application — 1 h 30
6. Module 5 : Synchronisation — 1 h 45
7. Module 6 : Versions et rollback — 1 h 15
8. Module 7 : Supervision et dépannage — 1 h 45
9. Module 8 : Déploiements bleu-vert et canari — 1 h
10. Module 9 : Helm et Kustomize — 1 h 30
11. Module 10 : Intégration continue avec ArgoCD — 1 h 15
12. Module 11 : Sécurité, identité et secrets — 1 h 15
13. Module 12 : Meilleures pratiques et scénarios réels — 1 h 30

🎓 Module 0 : Accueil et préparation

Durée
30 min (S0.1 : 30 min)

Prérequis
poste conforme à `setup-environnement.md`

Fil rouge
présentation d'*Hexa Mobilité*

Objectifs

- Situer les 3 jours de formation et leurs résultats attendus
- Décrire le fil rouge *Hexa Mobilité* et son état final visé
- Vérifier le poste de travail avec `preflight.sh` (0 KO)
- Évaluer son niveau de départ (auto-positionnement)

Déroulé

- Séquence S0.1 — Accueil, fil rouge et vérification des postes (30 min)

Séquence S0.1

Accueil, fil rouge et vérification des postes

30 min

1. ArgoCD en 3 jours
2. Le programme en un schéma
3. Les 6 parties du programme
4. Fil rouge : Hexa Mobilité
5. Fil rouge : état final visé
6. Environnement de travail
7. Vérifier son poste
8. Méthode de travail

📖 ArgoCD en 3 jours

Du cluster vide à la chaîne GitOps complète

```
Formation = 3 jours × 7 h = 21 h (+ 60 % de pratique)
```

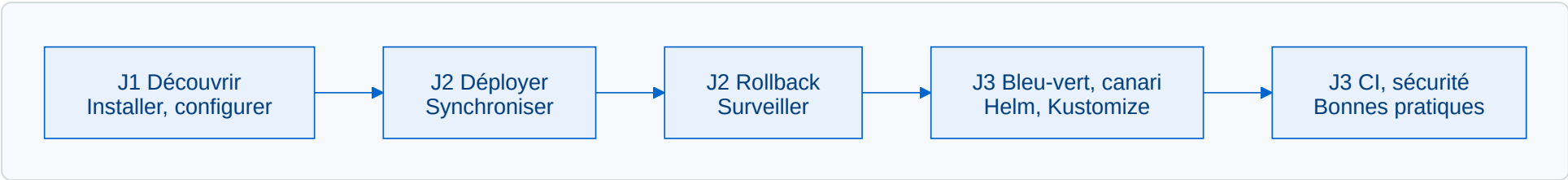
JOUR	THÈME	RÉSULTAT
J1	Introduction, installation, configuration	ArgoCD installé, accès sécurisés
J2	Déploiement et gestion des applications	Déployée, réversible, supervisée
J3	Avancé et bonnes pratiques	Canari, Helm, CI, sécurité, échelle

📌 Validation

QCM d'ancrage en fin de module, évaluation finale en fin de J3

Le programme en un schéma

Chaque module s'appuie sur le résultat du précédent



- Bilan QCM en fin de J1 et de J2
- Étude de cas et synthèse en fin de J3

📖 Les 6 parties du programme

Chaque partie du programme couverte par un ou plusieurs modules

PARTIE DU PROGRAMME	MODULES	JOUR
1. Introduction à ArgoCD	M00, M01	J1
2. Configuration d'ArgoCD	M02, M03	J1
3. Déploiement d'applications	M04, M05, M06	J2
4. Gestion des déploiements	M07, M08	J2, J3
5. Fonctionnalités avancées	M09, M10, M11	J3
6. Meilleures pratiques et cas réels	M12	J3

📖 Fil rouge : Hexa Mobilité

Éditeur fictif d'une application de suivi de flotte de véhicules

`flotte-web = frontend + API (image podinfo 6.15.0 → 6.17.0)`

🔄 Avant / Après

Avant: `kubectl apply` depuis les postes → aucune trace → écarts recette / prod
Après: commit → CI → dépôt de configuration → ArgoCD → cluster (audité, réversible)

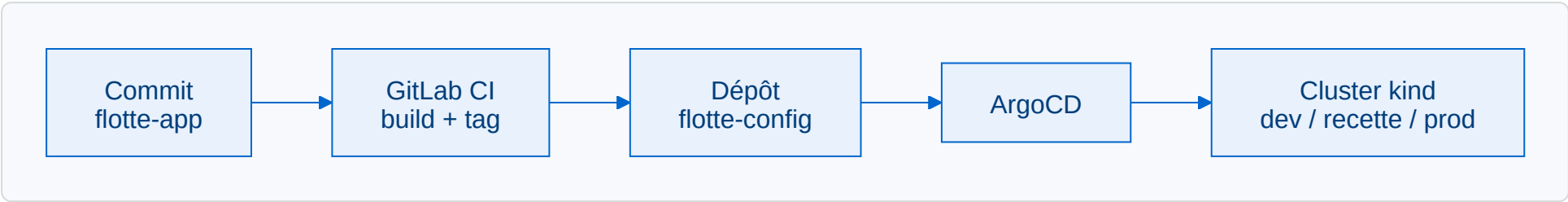
- Trois équipes : `dev` (déploie), `ops` (administre), `audit` (lecture seule)
- Trois environnements à terme : `dev`, `recette`, `prod`

🏢 En entreprise

point de départ fréquent (déploiements manuels, *drift* non détecté)

🕮 Fil rouge : état final visé

La chaîne construite au fil des modules



- `flotte-app` : code et pipeline CI (J3)
- `flotte-config` : manifestes, chart Helm, overlays Kustomize (J2 et J3)

📖 Environnement de travail

Un cluster par stagiaire, sur son propre poste

```
Poste = Docker + kind + kubectl + argocd + helm + git
```

OUTIL	VERSION	RÔLE
kind	v0.33.0	Cluster Kubernetes local
Kubernetes	v1.36.4	Cluster cible (nœud kind)
Argo CD	v3.6.1	Serveur et CLI
GitLab.com	SaaS	Dépôts et CI (compte personnel)

⚠ Piège courant

port 8080 déjà occupé → l'accès à l'UI échoue au J1

🕮 Vérifier son poste

Un contrôle par contrainte, sortie OK / KO

```bash
cd scripts/setup
./preflight.sh
```

Sortie attendue (extrait) :

```
OK    CPU: 8 vCPU (min 4)
OK    Container engine: docker
OK    kind version v0.33.0
OK    Port 8080 free
----
Result: workstation ready
```

Bonne pratique

lancer aussi `./pull-images.sh` (images téléchargées une seule fois)

📖 Méthode de travail

Théorie courte, pratique immédiate

- ⭐ Lab guidé : commandes fournies, résultat attendu indiqué
- ⭐⭐ Exercice semi-autonome : objectif + indices, sans les commandes
- ⭐⭐⭐ Défi autonome : problème ouvert (pour les plus rapides)
- 🚀 Bonus : extension facultative
- Corrections publiées après chaque lab

💡 Bonne pratique

un lab bloqué > 5 min → signaler (script de rattrapage disponible)

À retenir

Points clés

- 3 jours = Installer (J1) → Déployer et gérer (J2) → Industrialiser (J3)
- Fil rouge *Hexa Mobilité* : de `kubectl apply` manuel à GitOps complet
- Un cluster `kind` par poste : aucune ressource cloud, aucun coût
- `preflight.sh` = 0 KO avant le premier lab

QCM d'ancrage (1/3)

Questions 1 à 2

1. Quelle commande affiche les pods de tous les namespaces ?

- **A.** `kubectl get pods`
- **B.** `kubectl get pods -A`
- **C.** `kubectl describe namespaces`
- **D.** `kubectl logs --all`

Réponse B — `-A` (`--all-namespaces`) liste les pods de tous les namespaces ; sans option, seul le namespace courant est affiché.

Révéler la réponse

2. Quel objet Kubernetes maintient un nombre donné de pods identiques ?

- **A.** Service
- **B.** ConfigMap
- **C.** Deployment
- **D.** Ingress

Réponse C — Le Deployment gère un ReplicaSet qui maintient le nombre de réplicas demandé.

Révéler la réponse

QCM d'ancrage (2/3)

Questions 3 à 4

3. Quelle commande Git annule un commit en créant un nouveau commit inverse ?

- **A.** `git reset --hard HEAD~1`
- **B.** `git revert <COMMIT>`
- **C.** `git checkout <COMMIT>`
- **D.** `git stash`

Réponse B — `git revert` crée un commit inverse sans réécrire l'historique : geste clé du rollback GitOps (M06).

Révéler la réponse

4. Quel rôle joue un pipeline CI dans une chaîne de livraison ?

- **A.** Héberger le code source
- **B.** Construire, tester et publier un artefact à chaque modification
- **C.** Remplacer le cluster Kubernetes
- **D.** Stocker les secrets de production

Réponse B — La CI construit, teste et publie l'artefact (image) ; en GitOps, le déploiement est délégué à ArgoCD.

Révéler la réponse

QCM d'ancrage (3/3)

Questions 5 à 6

5. Quel outil sert à packager des applications Kubernetes avec des valeurs paramétrables ?

- **A.** Helm
- **B.** kind
- **C.** etcd
- **D.** kubeadm

Réponse A — Helm package les manifestes en charts paramétrables par `values.yaml` (vu au M09).

Révéler la réponse

6. Que signifie GitOps, en une phrase ?

- **A.** Héberger Git dans Kubernetes
- **B.** Git comme source de vérité de l'état souhaité, appliqué par un agent
- **C.** Lancer `git push` depuis le cluster
- **D.** Remplacer la CI par des hooks Git

Réponse B — GitOps : l'état souhaité est déclaré dans Git et un agent (ArgoCD) l'applique et le maintient en continu.

Révéler la réponse

🎓 Module 1 : GitOps et architecture d'ArgoCD

Durée
1 h (S1.1 : 30 min · S1.2 : 30 min)

Prérequis
bases Kubernetes et Git

Fil rouge
Hexa Mobilité déploie encore par `kubectl apply`

Objectifs

- Expliquer les 4 principes GitOps et la différence *push* / *pull*
- Identifier une dérive de configuration et son mécanisme de correction
- Placer les composants d'ArgoCD sur un schéma d'architecture
- Décrire le cycle d'un déploiement, de `git push` à `Synced` / `Healthy`

Déroulé

- Séquence S1.1 — GitOps : principes, push et pull (30 min)
- Séquence S1.2 — Architecture et composants d'ArgoCD (30 min)

Séquence S1.1

GitOps : principes, push et pull

30 min

1. Qu'est-ce que GitOps ?
2. Modèle push : la CI déploie
3. Modèle pull : le cluster tire
4. Push ou pull : comparaison
5. La boucle de réconciliation
6. Qu'est-ce qu'une dérive ?
7. Outils GitOps et CI/CD

Qu'est-ce que GitOps ?

Git comme source de vérité de l'état souhaité

GitOps = Déclaratif + Versionné + Tiré automatiquement + Réconcilié en continu

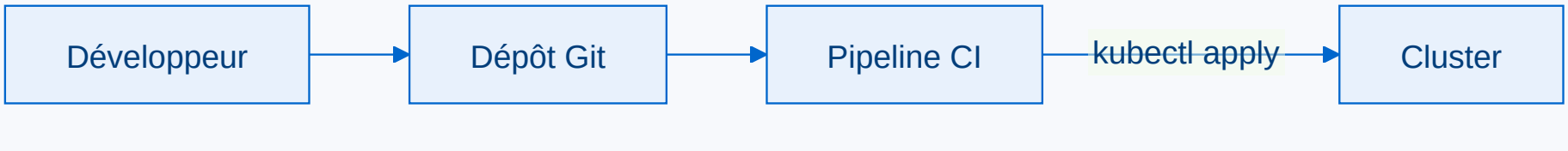
- Déclaratif : décrire l'état voulu (pas la suite de commandes pour l'obtenir)
- Versionné : état stocké dans Git (historique immuable, revue par merge request)
- Tiré : un agent dans le cluster récupère l'état depuis Git
- Réconcilié : l'agent compare et corrige en boucle (pas une seule fois)

Référence

4 principes définis par le projet OpenGitOps (CNCF), version 1.0.0

Modèle push : la CI déploie

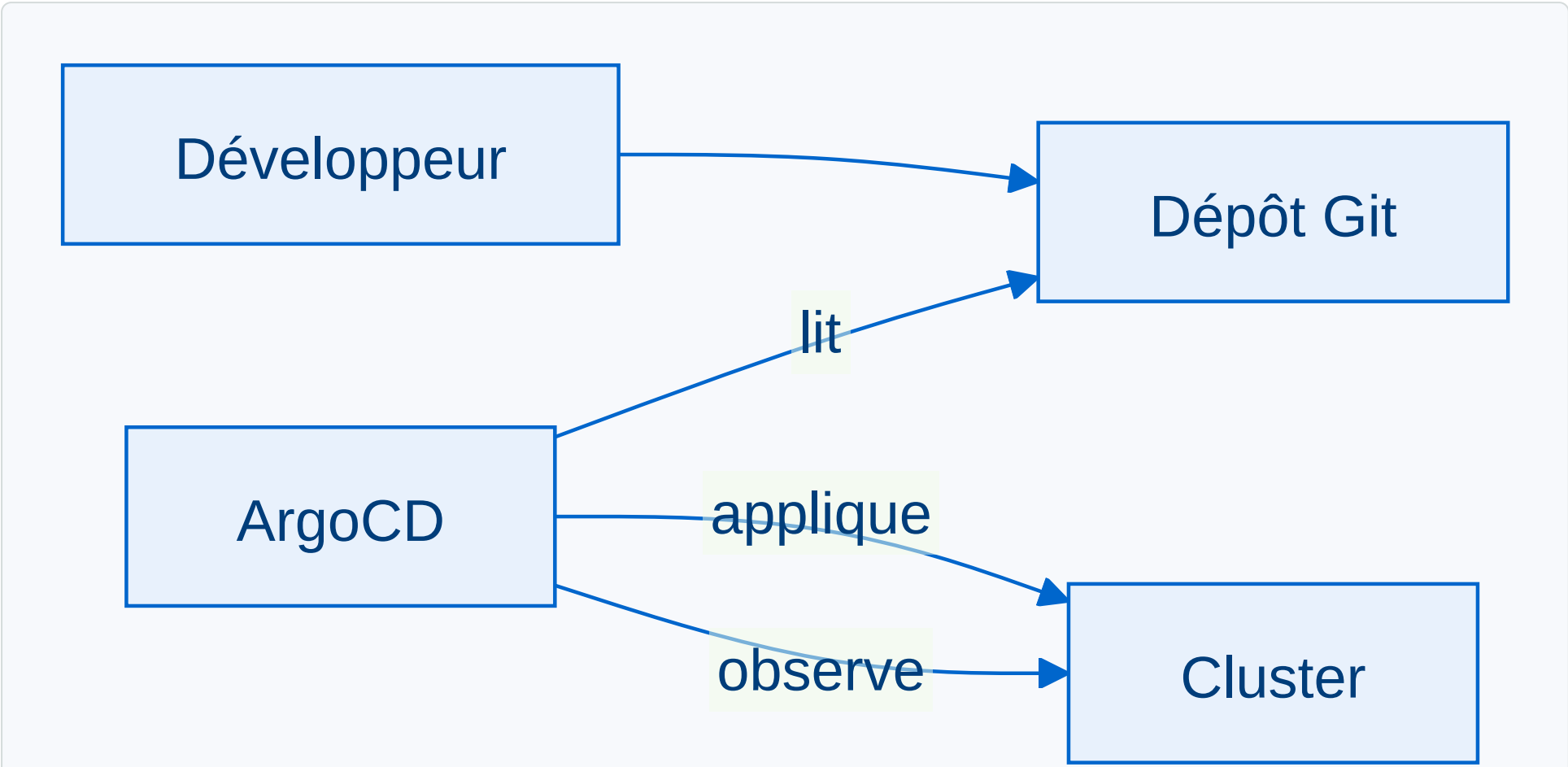
Le pipeline détient les accès au cluster



- Identifiants du cluster stockés dans la CI (surface d'attaque)
- Déploiement ponctuel : aucune correction si le cluster change ensuite
- Audit dispersé entre journaux CI et commandes manuelles

Modèle pull : le cluster tire

L'agent ArgoCD vit DANS le cluster



- Aucun identifiant du cluster hors du cluster

📖 Push ou pull : comparaison

Même objectif, garanties différentes

CRITÈRE	PUSH (CI)	PULL (GITOPS)
Accès au cluster	Depuis la CI	Depuis le cluster
Dérive détectée	Non	Oui, en continu
Retour arrière	Relancer un ancien pipeline	`git revert`
Audit	Journaux CI	Historique Git
Cluster reconstruit	Rejouer les pipelines	Pointer ArgoCD sur Git

En entreprise

CI et GitOps coexistent (la CI construit l'image, ArgoCD déploie)

La boucle de réconciliation

Observer, comparer, agir, recommencer

Écart = État souhaité (Git) − État réel (cluster)



- Intervalle par défaut : 120 s + jitter jusqu'à 60 s (`timeout.reconciliation`)
- Webhook Git : détection immédiate d'un nouveau commit

❓ Qu'est-ce qu'une dérive ?

Le cluster ne correspond plus à Git

Dérive (drift) = modification hors Git

🔄 Avant / Après

Avant: `kubectl scale --replicas=5` en urgence → oublié → recette ≠ prod
Après: ArgoCD affiche `OutOfSync` → self-heal remet 2 réplicas (valeur de Git)

- Causes : correctif manuel, opérateur tiers, script, console cloud
- Réponses : alerte seule, ou correction automatique (*self-heal*, M05)

⚠️ Piège courant

self-heal actif + correctif manuel en urgence = correctif annulé

∞ Outils GitOps et CI/CD

ArgoCD dans l'écosystème

OUTIL	RÔLE	MODÈLE
Argo CD	CD GitOps Kubernetes, UI riche	Pull
Flux	CD GitOps Kubernetes, sans UI native	Pull
GitLab CI, Jenkins	Construire, tester, publier	Push
Helm, Kustomize	Générer les manifestes	Outils de rendu

- Argo CD : projet CNCF diplômé (*graduated*, 2022)

Séquence S1.2

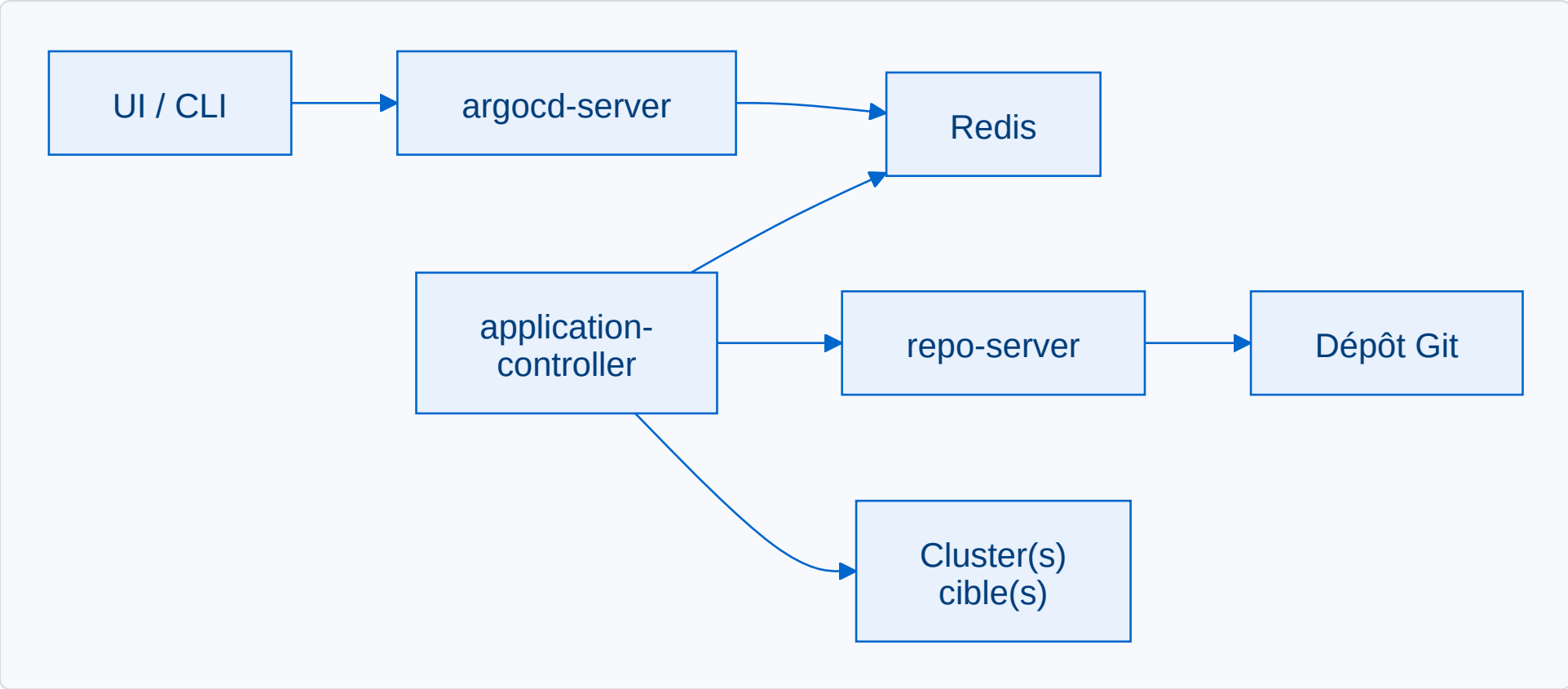
Architecture et composants d'ArgoCD

30 min

1. ArgoCD en un schéma
2. Les composants (1/2)
3. Les composants (2/2)
4. Du commit au déploiement
5. Deux statuts à lire
6. Les ressources ArgoCD
7. Caractéristiques et avantages d'ArgoCD
8. Quoi de neuf depuis ArgoCD 3.0 ?

🕮 ArgoCD en un schéma

Trois composants cœur, un namespace dédié (`argocd`)



- `repo-server` : Git → manifestes ; `application-controller` : manifestes → cluster

🕮 Les composants (1/2)

Le cœur du moteur

COMPOSANT	TYPE	RÔLE
`argocd-server`	Deployment	API gRPC/REST, UI, authentification
`argocd-repo-server`	Deployment	Clone Git, rendu Helm / Kustomize / YAML
`argocd-application-controller`	StatefulSet	Compare, synchronise, calcule la santé

- Contrôleur = StatefulSet (sharding par cluster possible, M12)
- `repo-server` : aucun accès à l'API Kubernetes (rendu uniquement)

📖 Les composants (2/2)

Services complémentaires

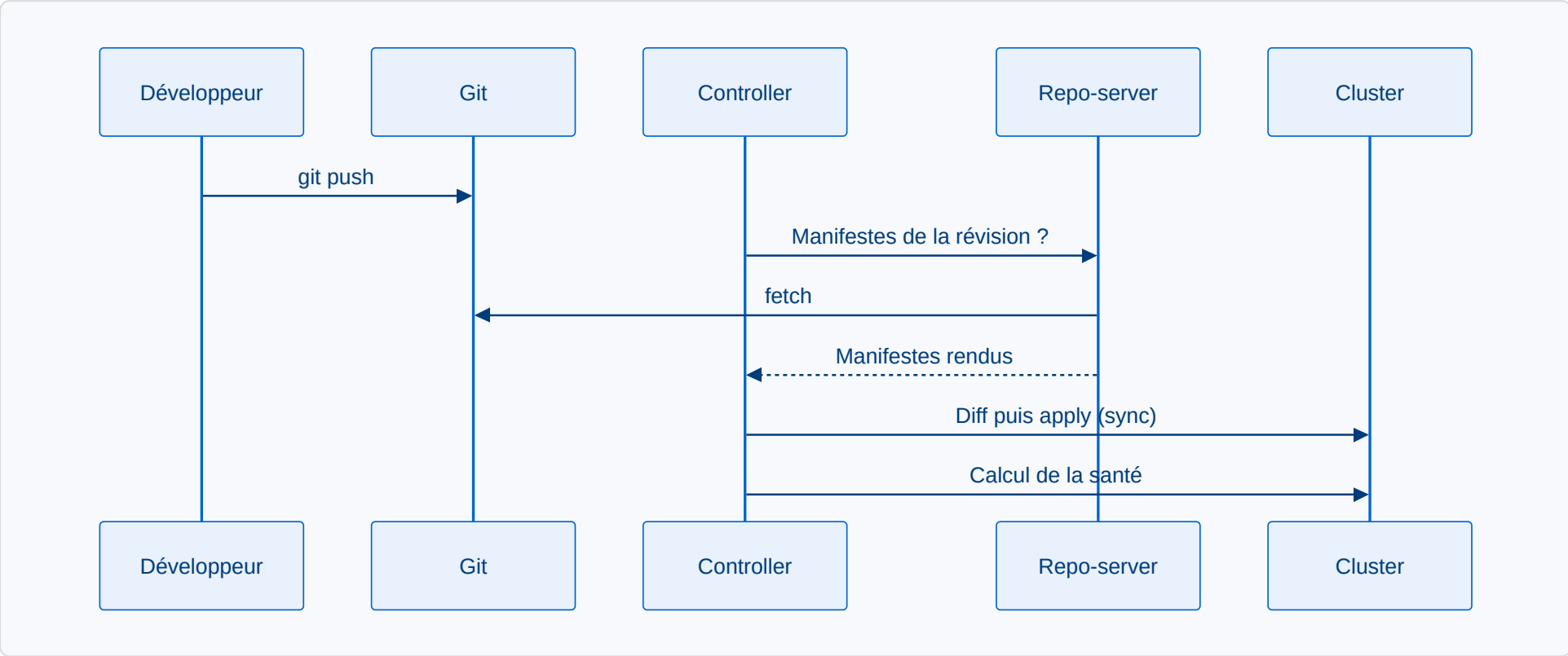
COMPOSANT	RÔLE
`argocd-applicationset-controller`	Génère des Applications en série (M12)
`argocd-notifications-controller`	Alertes Slack, mail, webhook (M07)
`argocd-redis`	Cache des manifestes et de l'état
`argocd-dex-server`	SSO : OIDC, LDAP, SAML, GitHub, GitLab (M03)

💡 Bonne pratique

Redis = cache uniquement (perte = recalcul, pas de perte de données)

∞ Du commit au déploiement

Qui fait quoi, dans quel ordre



Deux statuts à lire

Synchronisation ET santé : deux questions distinctes

STATUT	VALEURS	QUESTION
Sync	`Synced`, `OutOfSync`, `Unknown`	Le cluster = Git ?
Health	`Healthy`, `Progressing`, `Degraded`	L'application fonctionne ?
Health (suite)	`Suspended`, `Missing`, `Unknown`	

- `Synced` + `Degraded` : Git appliqué, mais application en échec (image, config)
- `OutOfSync` + `Healthy` : application fonctionnelle, mais différente de Git

📖 Les ressources ArgoCD

Tout est déclaratif, y compris ArgoCD lui-même

```
Application = Source (Git) + Destination (cluster, namespace) + Projet
```

CRD	RÔLE	MODULE
`Application`	Une application déployée depuis Git	M04
`AppProject`	Périmètre : dépôts, clusters, droits	M03
`ApplicationSet`	Fabrique d'Applications	M12

- Configuration globale : ConfigMaps `argocd-cm`, `argocd-rbac-cm` (M03)

🕮 Caractéristiques et avantages d'ArgoCD

Ce que l'outil apporte, au-delà du principe GitOps

CARACTÉRISTIQUE	AVANTAGE
UI web, CLI, API	Visualisation de l'état en temps réel, automatisation
Helm, Kustomize, YAML, plugins	Formats existants conservés
Multi-clusters, AppProjects	Une instance pour plusieurs équipes et environnements
SSO, RBAC	Accès alignés sur l'annuaire de l'entreprise
Hooks, vagues, rollback	Livraisons ordonnées et réversibles
Notifications, métriques	Pannes connues avant les utilisateurs

❓ Quoi de neuf depuis ArgoCD 3.0 ?

Changements à connaître en reprenant d'anciens exemples (2.x)

- Suivi des ressources par annotation par défaut (et non plus par label)
- RBAC fin : `update` / `delete` ne s'appliquent plus aux sous-ressources
- Accès aux journaux (`logs`) soumis au RBAC
- Dépôts déclarés dans `argocd-cm` : syntaxe historique supprimée (Secrets)
- Métriques `argocd_app_sync_status` et `argocd_app_health_status` retirées

⚠ Piège courant

tutoriels 2.x encore très présents en ligne (vérifier la version)

À retenir

Points clés

- GitOps = Git source de vérité + agent qui tire + réconciliation continue
- Pull : identifiants du cluster jamais hors du cluster, dérive détectée
- `repo-server` rend les manifestes, `application-controller` les applique
- Deux statuts indépendants : Sync (= Git ?) et Health (fonctionne ?)
- Exemples à lire en version 3.x (annotations, RBAC fin, dépôts en Secrets)

QCM d'ancrage (1/4)

Questions 1 à 2

1. Quel principe GitOps garantit la correction d'une modification manuelle ?

- **A.** Déclaratif
- **B.** Versionné et immuable
- **C.** Tiré automatiquement
- **D.** Réconcilié en continu

Réponse D — La réconciliation continue compare Git et cluster en boucle et corrige l'écart (si self-heal actif).

Révéler la réponse

2. Dans le modèle pull, où se trouvent les identifiants d'accès au cluster ?

- **A.** Dans les variables de la CI
- **B.** Dans le cluster, utilisés par ArgoCD
- **C.** Dans le dépôt Git, chiffrés
- **D.** Sur le poste du développeur

Réponse B — En pull, l'agent est dans le cluster ; aucun identifiant du cluster n'est exposé à l'extérieur.

Révéler la réponse

QCM d'ancrage (2/4)

Questions 3 à 4

3. Quel composant clone le dépôt Git et génère les manifestes ?

- **A.** `argocd-server`
- **B.** `argocd-application-controller`
- **C.** `argocd-repo-server`
- **D.** `argocd-redis`

Réponse C — `argocd-repo-server` clone le dépôt et rend les manifestes (YAML, Helm, Kustomize).

Révéler la réponse

4. Quel composant applique les manifestes sur le cluster ?

- **A.** `argocd-application-controller`
- **B.** `argocd-repo-server`
- **C.** `argocd-dex-server`
- **D.** `argocd-notifications-controller`

Réponse A — Seul `argocd-application-controller` applique les ressources sur les clusters cibles.

Révéler la réponse

QCM d'ancrage (3/4)

Questions 5 à 6

5. Une application est `Synced` et `Degraded`. Que conclure ?

- **A.** Le cluster diffère de Git
- **B.** ArgoCD est en panne
- **C.** Git est appliqué, mais l'application ne fonctionne pas
- **D.** La synchronisation est en cours

Réponse C — Sync et Health sont indépendants : Git est appliqué mais les pods échouent (image, configuration).

Révéler la réponse

6. Sans webhook, après quel délai maximal par défaut un commit est-il détecté ?

- **A.** 10 secondes
- **B.** Environ 3 minutes (120 s + jitter de 60 s)
- **C.** 15 minutes
- **D.** Jamais, une synchronisation manuelle est requise

Réponse B — `timeout.reconciliation` vaut 120 s par défaut, avec un jitter jusqu'à 60 s.

Révéler la réponse

QCM d'ancrage (4/4)

Questions 7 à 8

7. Quelle ressource définit les dépôts et clusters autorisés pour une équipe ?

- **A.** `Application`
- **B.** `ApplicationSet`
- **C.** `AppProject`
- **D.** `argocd-cm`

Réponse C — L'`AppProject` restreint dépôts sources, destinations et types de ressources.

Révéler la réponse

8. Depuis ArgoCD 3.0, comment ArgoCD suit-il par défaut les ressources qu'il gère ?

- **A.** Par label `app.kubernetes.io/instance`
- **B.** Par annotation
- **C.** Par nom de namespace
- **D.** Par propriétaire (`ownerReferences`)

Réponse B — Depuis la 3.0, le suivi par annotation (`argocd.argoproj.io/tracking-id`) est le défaut.

Révéler la réponse

Module 2 : Installation d'ArgoCD sur Kubernetes

Durée

1 h 45 (S2.1 : 1 h · S2.2 : 45 min)

Prérequis

`preflight.sh` sans KO

Fil rouge

Hexa Mobilité prépare sa plateforme GitOps

Objectifs

- Créer un cluster `kind` et y installer ArgoCD v3.6.1
- Vérifier l'état de tous les composants (pods `Running`)
- Accéder à l'UI et à la CLI, changer le mot de passe `admin`
- Exposer l'UI sans `port-forward` (NodePort)

Déroulé

- Séquence S2.1 — Cluster kind et installation d'ArgoCD (1 h)
- Séquence S2.2 — Accès à l'UI et à la CLI (45 min)

Séquence S2.1

Cluster kind et installation d'ArgoCD

1 h

1. Quelle installation choisir ?
2. L'environnement du lab
3. Créer le cluster
4. Installer ArgoCD
5. Pourquoi `--server-side` ?
6. Vérifier l'installation
7. Installer pour la production

❓ Quelle installation choisir ?

Un manifeste officiel par besoin

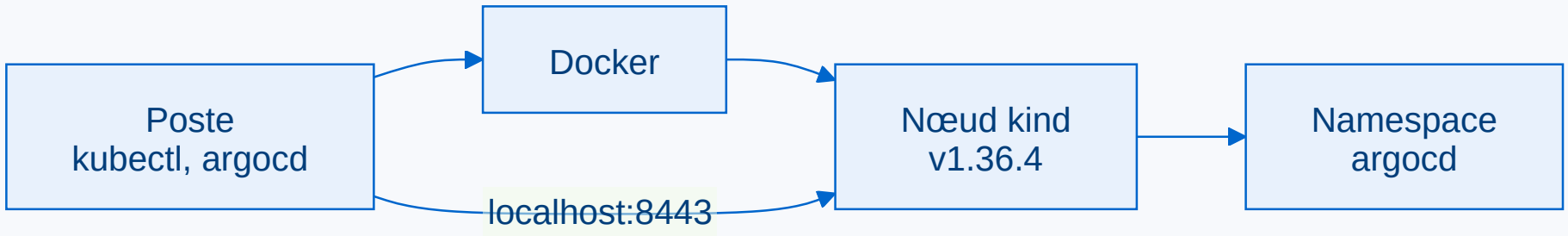
`Installation = Manifeste (version figée) + Namespace argocd`

VARIANTE	FICHIER OU OUTIL	USAGE
Standard	`manifests/install.yaml`	Formation, petites équipes
Haute disponibilité	`manifests/ha/install.yaml`	Production (3 nœuds minimum)
Core	`manifests/core-install.yaml`	Sans UI ni SSO (CLI `--core`)
Namespaced	`manifests/namespace-install.yaml`	Droits limités à des namespaces
Chart Helm	`argo/argo-cd` (argo-helm)	Valeurs paramétrables

- Version figée (`v3.6.1`), jamais la branche `stable` en production

L'environnement du lab

Un cluster Kubernetes complet dans un conteneur



- Nœud = conteneur `argocd-formation-control-plane`
- Port 8443 du poste → NodePort 30443 du nœud (exercice 02.3)

📖 Créer le cluster

Configuration déclarative, versionnée dans le dépôt

```bash
kind create cluster --config scripts/setup/kind-config.yaml
kubectl get nodes
```

Sortie attendue :

```
NAME                              STATUS   ROLES           AGE   VERSION
argocd-formation-control-plane    Ready    control-plane   40s   v1.36.4
```

⚠ Piège courant

port 8443 déjà utilisé sur le poste → création du cluster refusée

Installer ArgoCD

Un namespace, un manifeste, une commande

```bash
kubectl create namespace argocd
kubectl apply -n argocd --server-side --force-conflicts \
  -f https://raw.githubusercontent.com/argoproj/argo-cd/v3.6.1/manifests/install.yaml
```

Sortie attendue (fin) :

```
networkpolicy.networking.k8s.io/argocd-server-network-policy serverside-applied
```

- Contenu : 3 CRD, 6 Deployments, 1 StatefulSet, RBAC, NetworkPolicies (voir M01)

❓ Pourquoi `--server-side` ?

Exigence de la documentation officielle 3.x

`Apply client = annotation last-applied (limite 262 Ko)`

- CRD `ApplicationSet` > 262 Ko : `kubectl apply` classique échoue
- `--server-side` : calcul de la fusion par l'API server, sans annotation
- `--force-conflicts` : reprise des champs gérés par un autre outil (mise à jour)

🔄 Avant / Après

Avant (2.x): `kubectl apply -n argocd -f install.yaml`
Après (3.x): `kubectl apply -n argocd --server-side --force-conflicts -f install.yaml`

🕮 Vérifier l'installation

7 pods attendus, tous `Running`

```bash
kubectl wait --for=condition=Available deployment --all -n argocd --timeout=600s
kubectl get pods -n argocd
```

Sortie attendue :

```
NAME                                                READY   STATUS    RESTARTS   AGE
argocd-application-controller-0                     1/1     Running   0          34s
argocd-applicationset-controller-f6bc6968d-xtpwx    1/1     Running   0          35s
argocd-dex-server-55b9777447-tsv7g                  1/1     Running   0          35s
argocd-notifications-controller-b9499f64b-jnzxh     1/1     Running   0          35s
argocd-redis-79f897d5d9-cp6lp                       1/1     Running   0          35s
argocd-repo-server-75fcf89b6b-wdm57                 1/1     Running   0          35s
argocd-server-67795db55d-pzk69                      1/1     Running   0          34s
```

Installer pour la production

Au-delà du lab

- Version figée + mise à jour planifiée (notes de version `upgrading/3.x-3.y`)
- Manifeste HA : réplicas multiples, Redis HA, anti-affinité (3 nœuds minimum)
- Personnalisation par Kustomize ou Helm, jamais en modifiant `install.yaml`
- ArgoCD géré par ArgoCD lui-même (*self-managed*, M12)
- Accès par Ingress ou Gateway API + certificat TLS reconnu

En entreprise

`kustomization.yaml` = `install.yaml` v3.6.1 + patchs maison (défi 02.4)

Séquence S2.2

Accès à l'UI et à la CLI

45 min

1. Comment accéder à ArgoCD ?
2. Premier accès : port-forward
3. Se connecter avec la CLI
4. Le tableau de bord en un schéma
5. Premier déploiement : guestbook
6. Lire la vue d'une application
7. Sécuriser après l'installation

Comment accéder à ArgoCD ?

Le service `argocd-server` n'est pas exposé par défaut

MÉTHODE	COMMANDE OU OBJET	USAGE
Port-forward	`kubectl port-forward`	Test, poste local
NodePort	`type: NodePort`	Lab, cluster sans load balancer
LoadBalancer	`type: LoadBalancer`	Cloud (IP externe)
Ingress / Gateway	`Ingress`, `HTTPRoute`	Production (nom DNS, TLS)

Ingress

fin de maintenance d'ingress-nginx annoncée (mars 2026) → Gateway API à privilégier

📖 Premier accès : port-forward

Tunnel temporaire du poste vers le service

```bash
kubectl port-forward svc/argocd-server -n argocd 8080:443
```

Sortie attendue :

```
Forwarding from 127.0.0.1:8080 -> 8080
Forwarding from [::1]:8080 -> 8080
```

- UI : `https://localhost:8080` (certificat auto-signé à accepter)
- Terminal bloqué : ouvrir un second terminal pour la suite

🕮 Se connecter avec la CLI

Mot de passe initial généré à l'installation

```bash
argocd admin initial-password -n argocd
argocd login localhost:8080 --username admin --insecure
argocd account update-password
```

Sortie attendue :

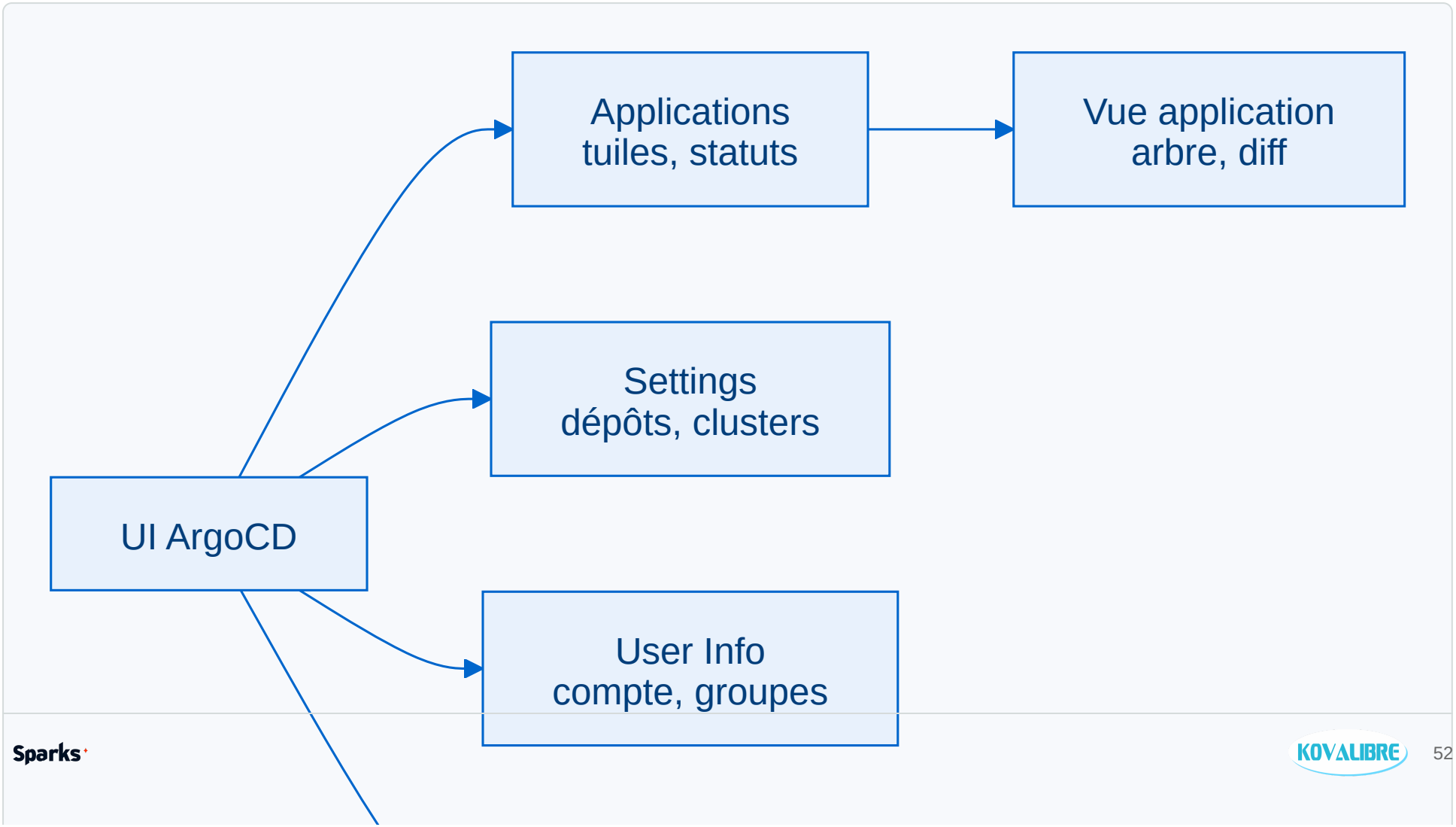
```
'admin:login' logged in successfully
Context 'localhost:8080' updated
Password updated
```

⚠ Piège courant

`--insecure` obligatoire (certificat auto-signé) ; à proscrire en production

🕮 Le tableau de bord en un schéma

Menu latéral : 4 entrées principales



∞ Premier déploiement : guestbook

Application d'exemple officielle (`argocd-example-apps`)

```bash
argocd app create guestbook \
  --repo https://github.com/argoproj/argocd-example-apps.git --path guestbook \
  --dest-server https://kubernetes.default.svc --dest-namespace default
argocd app sync guestbook
argocd app list
```

Sortie attendue (colonnes principales) :

```
NAME              CLUSTER                         NAMESPACE  PROJECT  STATUS  HEALTH
argocd/guestbook  https://kubernetes.default.svc  default    default  Synced  Healthy
```

📖 Lire la vue d'une application

Tout l'état d'une application sur un écran

ZONE	CONTENU	ACTION
En-tête	Sync, Health, révision Git	*Sync*, *Refresh*
Arbre	Application → Service, Deployment → ReplicaSet → Pod	Clic : manifeste, événements, logs
App diff	Écart Git ↔ cluster	Avant une sync
History and rollback	Révisions déployées	M06

- Vues alternatives : réseau, liste, pods (icônes en haut à droite)

📖 Sécuriser après l'installation

Trois gestes immédiats

- Changer le mot de passe `admin` (`argocd account update-password`)
- Supprimer le secret initial (`kubectl delete secret argocd-initial-admin-secret -n argocd`)
- Créer des comptes nominatifs puis désactiver `admin` (M03)

```bash
kubectl get secret argocd-initial-admin-secret -n argocd
```

Sortie attendue (après suppression) :

```
Error from server (NotFound): secrets "argocd-initial-admin-secret" not found
```

💡 Bonne pratique

secret initial = mot de passe en clair, inutile après le premier changement

À retenir

Points clés

- Installation = manifeste versionné + `kubectl apply --server-side --force-conflicts`
- 7 pods dans le namespace `argocd`, vérifiés par `kubectl wait`
- Accès : port-forward (test) → NodePort (lab) → Ingress / Gateway + TLS (production)
- `argocd login` + changement de mot de passe + suppression du secret initial
- UI : Applications (statuts, arbre, diff, historique) et Settings (dépôts, clusters, projets)

QCM d'ancrage (1/4)

Questions 1 à 2

1. Pourquoi l'option `--server-side` est-elle requise pour installer ArgoCD 3.x ?

- **A.** Pour installer plus vite
- **B.** Parce qu'une CRD dépasse la taille maximale de l'annotation `last-applied`
- **C.** Pour activer TLS
- **D.** Pour installer dans un autre namespace que `argocd`

Réponse B — La CRD `ApplicationSet` dépasse 262 Ko, taille maximale de l'annotation `last-applied-configuration` de l'apply client.

Révéler la réponse

2. Quel manifeste choisir pour une production sur 3 nœuds ?

- **A.** `install.yaml`
- **B.** `core-install.yaml`
- **C.** `ha/install.yaml`
- **D.** `namespace-install.yaml`

Réponse C — `ha/install.yaml` : réplicas multiples, Redis HA, anti-affinité exigeant au moins 3 nœuds.

Révéler la réponse

QCM d'ancrage (2/4)

Questions 3 à 4

3. Où se trouve le mot de passe initial du compte `admin` ?

- **A.** Dans la ConfigMap `argocd-cm`
- **B.** Dans le secret `argocd-initial-admin-secret`
- **C.** Dans les journaux de `argocd-server`
- **D.** Il vaut `admin` par défaut

Réponse B — Le secret `argocd-initial-admin-secret`, à supprimer après changement du mot de passe.

Révéler la réponse

4. Quel composant porte le pod `argocd-application-controller-0` ?

- **A.** Un Deployment
- **B.** Un DaemonSet
- **C.** Un StatefulSet
- **D.** Un Job

Réponse C — Le contrôleur est un StatefulSet (nom de pod suffixé `-0`), pour permettre le sharding.

Révéler la réponse

QCM d'ancrage (3/4)

Questions 5 à 6

5. Quelle méthode d'accès convient à un cluster de production ?

- **A.** `kubectl port-forward`
- **B.** NodePort sur le poste formateur
- **C.** Ingress ou Gateway API avec certificat TLS reconnu
- **D.** Accès direct au pod `argocd-server`

Réponse C — Ingress ou Gateway API avec certificat reconnu : nom DNS stable, TLS valide, pas de tunnel.

Révéler la réponse

6. Pourquoi l'option `--insecure` est-elle utilisée avec `argocd login` en lab ?

- **A.** Pour désactiver l'authentification
- **B.** Pour accepter le certificat auto-signé du serveur
- **C.** Pour utiliser HTTP au lieu de gRPC
- **D.** Pour se connecter sans mot de passe

Réponse B — `--insecure` désactive la vérification du certificat TLS, auto-signé en lab.

Révéler la réponse

QCM d'ancrage (4/4)

Questions 7 à 7

7. Une application vient d'être créée sans synchronisation. Quel statut affiche-t-elle ?

A. `Synced` / `Healthy`

B. `OutOfSync` / `Missing`

C. `Synced` / `Degraded`

D. `Unknown` / `Progressing`

Réponse B — Sans synchronisation, les ressources n'existent pas encore : `OutOfSync` / `Missing`.

Révéler la réponse

Module 3 : Configuration et sécurité des accès

Durée

3 h (S3.1 : 45 min · S3.2 : 1 h · S3.3 : 1 h · bilan J1 : 15 min)

Prérequis

M02 terminé (`https://localhost:8443`)

Fil rouge

trois équipes à faire entrer dans ArgoCD

Objectifs

- Modifier les paramètres d'ArgoCD sans écraser la configuration par défaut
- Créer des comptes locaux et leur attribuer des rôles RBAC
- Isoler une équipe dans un AppProject (dépôts, destinations, ressources)
- Déclarer un dépôt Git privé et lister les clusters gérés

Déroulé

- Séquence S3.1 — Paramètres d'ArgoCD (45 min)
- Séquence S3.2 — Comptes, RBAC et rôles (1 h)
- Séquence S3.3 — AppProjects, dépôts et clusters (1 h)

Séquence S3.1

Paramètres d'ArgoCD

45 min

1. Où se configure ArgoCD ?
2. Paramètres courants de `argocd-cm`
3. Modifier sans écraser
4. Paramètres de démarrage

❓ Où se configure ArgoCD ?

Tout est déclaratif : ConfigMaps et Secrets du namespace `argocd`

`Configuration = argocd-cm + argocd-cmd-params-cm + argocd-rbac-cm + Secrets`

OBJET	CONTENU	PRISE EN COMPTE
`argocd-cm`	URL, comptes, UI, exclusions, réconciliation	À chaud (sauf exceptions)
`argocd-cmd-params-cm`	Paramètres de démarrage des composants	Redémarrage des pods
`argocd-rbac-cm`	Politique RBAC	À chaud
`argocd-secret`	Mots de passe, clé de signature des sessions	À chaud
Secrets étiquetés	Dépôts, clusters	À chaud

🕮 Paramètres courants de `argocd-cm`

Une clé = un comportement

CLÉ	EXEMPLE	EFFET
`url`	`https://localhost:8443`	URL publique (SSO, notifications, liens)
`ui.bannercontent`	`Cluster de formation`	Bannière en haut de l'UI
`statusbadge.enabled`	`"true"`	Badge de statut intégrable (README)
`admin.enabled`	`"false"`	Désactive le compte `admin`
`timeout.reconciliation`	`120s`	Intervalle de comparaison à Git
`exec.enabled`	`"true"`	Terminal web dans les pods (désactivé par défaut)

📖 Modifier sans écraser

`argocd-cm` n'est PAS vide en 3.x

- Contenu par défaut : `resource.exclusions`, `resource.customizations.ignoreResourceUpdates.*`
- `kubectl apply` d'un fichier partiel = valeurs par défaut supprimées

```bash
kubectl patch cm argocd-cm -n argocd --type merge -p \
  '{"data":{"ui.bannercontent":"Cluster de formation - Hexa Mobilite"}}'
kubectl get cm argocd-cm -n argocd -o jsonpath='{.data.ui\.bannercontent}{"\n"}'
```

Sortie attendue :

```
configmap/argocd-cm patched
Cluster de formation - Hexa Mobilite
```

💡 Bonne pratique

configuration d'ArgoCD versionnée (patchs Kustomize, défi 02.4)

📖 Paramètres de démarrage

`argocd-cmd-params-cm` : options des binaires, vide par défaut

CLÉ	EFFET
`server.insecure`	`argocd-server` en HTTP (TLS terminé par l'Ingress)
`server.basehref`	UI servie sous un sous-chemin (`/argocd`)
`controller.status.processors`	Parallélisme du contrôleur (nombreuses applications)
`reposerver.parallelism.limit`	Rendus simultanés du repo-server (mémoire)

⚠ Piège courant

modification sans `kubectl rollout restart` du composant = aucun effet

Séquence S3.2

Comptes, RBAC et rôles

1 h

1. Qui accède à ArgoCD ?
2. Créer des comptes locaux
3. Le modèle RBAC
4. Ressources et actions
5. La politique d'Hexa Mobilité
6. Tester une politique
7. Jetons API pour la CI

❓ Qui accède à ArgoCD ?

Trois sources d'identité

```
Identité = Compte local | Utilisateur SSO (Dex, OIDC) | Jeton API
```

SOURCE	DÉCLARATION	USAGE
Compte local	`accounts.<nom>` dans `argocd-cm`	Petites équipes, formation
SSO	Dex ou OIDC dans `argocd-cm`	Entreprise (groupes de l'annuaire)
Jeton API	Capacité `apiKey`	Pipelines CI (M10)

- Capacités d'un compte : `login` (UI, CLI) et / ou `apiKey` (jetons)

📖 Créer des comptes locaux

Déclaration dans `argocd-cm`, mot de passe par la CLI

```yaml
data:
  accounts.dev-alice: login
  accounts.ops-bob: login
  accounts.audit-claire: login
  accounts.ci-gitlab: apiKey
```

```bash
kubectl patch cm argocd-cm -n argocd --type merge --patch-file argocd-cm-accounts.yaml
argocd account update-password --account dev-alice --current-password '<ADMIN_PASSWORD>'
argocd account list
```

Sortie attendue (extrait) :

```
NAME          ENABLED  CAPABILITIES
ci-gitlab     true     apiKey
dev-alice     true     login
```

Le modèle RBAC

Qui peut faire quoi, sur quel objet

Règle = sujet + ressource + action + objet + effet

```
p, role:dev, applications, sync, flotte/*, allow
g, dev-alice, role:dev
```

- `p` (policy) : une permission accordée à un rôle
- `g` (grouping) : un utilisateur ou un groupe SSO reçoit un rôle
- Objet : `<projet>/<application>` pour applications, applicationsets, logs, exec
- Rôles intégrés : `role:readonly`, `role:admin`

⚠ Piège courant

`policy.default: role:readonly` = tout le monde voit tout

🕮 Ressources et actions

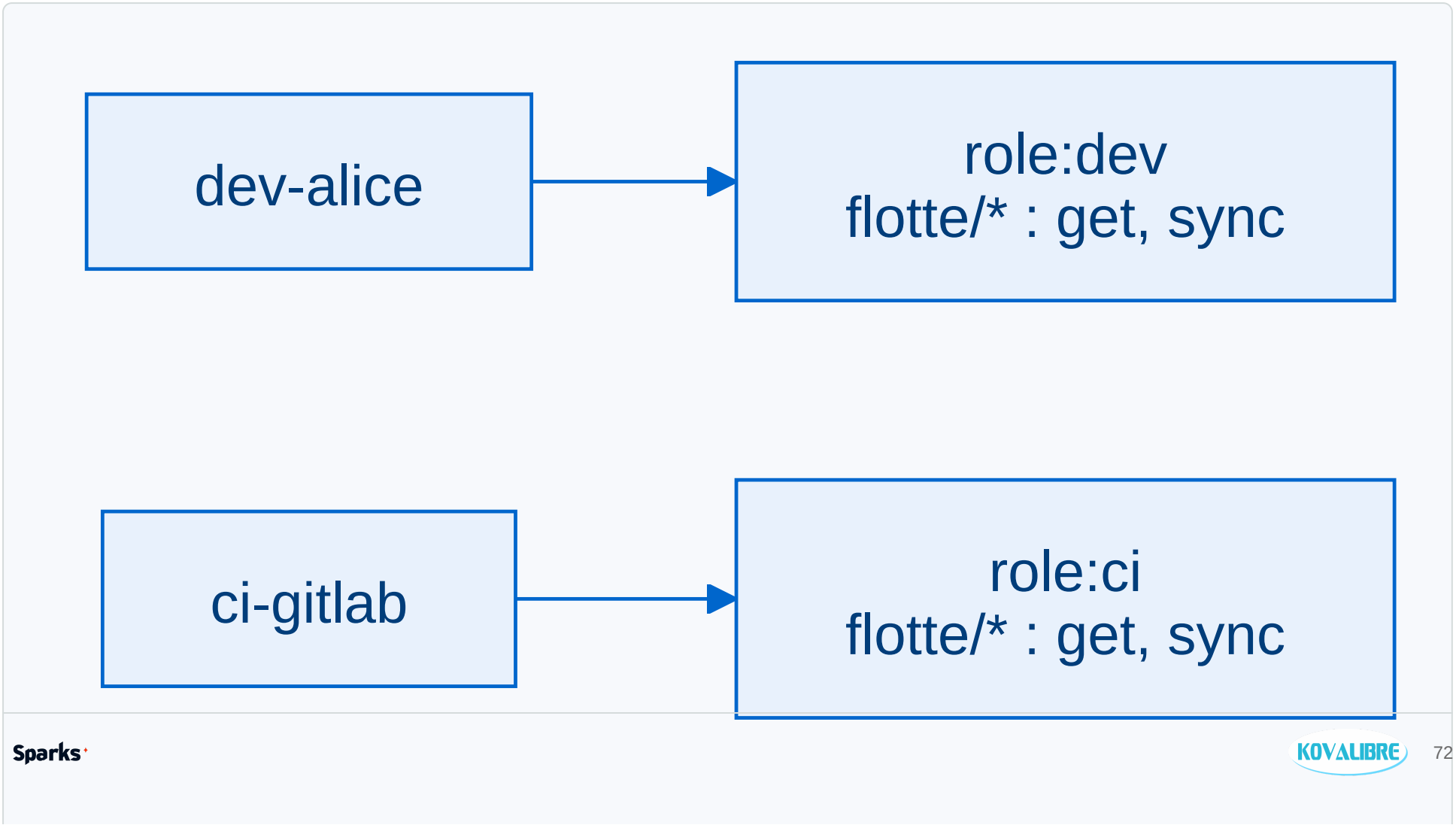
Vocabulaire de `argocd-rbac-cm`

RESSOURCE	ACTIONS PRINCIPALES
`applications`	`get`, `create`, `update`, `delete`, `sync`, `rollback`, `override`, `action/*`
`logs`, `exec`	`get` / `create` (soumis au RBAC depuis 3.0)
`projects`, `repositories`, `clusters`	`get`, `create`, `update`, `delete`
`accounts`	`get`, `update`

- 3.x : `update` / `delete` ne couvrent plus les sous-ressources (`update/*`, `delete/*`)
- `rollback` distincte de `sync` (M06)

🕮 La politique d'Hexa Mobilité

Trois équipes, un pipeline, aucun droit par défaut (`policy.default` vide)



🕮 Tester une politique

Avant de l'appliquer, puis connecté en tant qu'utilisateur

```bash
argocd admin settings rbac validate --policy-file argocd-rbac-cm.yaml
argocd admin settings rbac can dev-alice delete applications 'flotte/flotte-web' -n argocd
argocd login localhost:8443 --username dev-alice --insecure
argocd account can-i sync applications 'flotte/flotte-web'
```

Sortie attendue :

```
Policy is valid.
No
'dev-alice:login' logged in successfully
yes
```

`validate` ignore la politique intégrée → warning `roles not found` normal pour `role:admin` / `role:readonly`

💡 Bonne pratique

politique dans Git + `rbac validate` en CI avant fusion

Jetons API pour la CI

Un compte `apiKey` ne se connecte pas : il reçoit des jetons

```bash
argocd account generate-token --account ci-gitlab --expires-in 168h --id formation
argocd account get --account ci-gitlab
```

Sortie attendue (fin) :

```
Tokens:
ID         ISSUED AT                  EXPIRING AT
formation  2026-11-08T13:14:48+01:00  2026-11-15T13:14:48+01:00
```

- Jeton = secret : variable masquée et protégée dans la CI (M10)
- Révocation : `argocd account delete-token --account ci-gitlab formation`

Séquence S3.3

AppProjects, dépôts et clusters

1 h

1. Qu'est-ce qu'un AppProject ?
2. Le projet `flotte`
3. Le projet en action
4. Déclarer un dépôt privé
5. Dépôts : vérifier la connexion
6. Les clusters gérés

❓ Qu'est-ce qu'un AppProject ?

Une frontière autour d'une équipe

`AppProject = Dépôts sources + Destinations + Ressources autorisées + Rôles`

CHAMP	RÔLE
`sourceRepos`	Dépôts Git ou Helm autorisés
`destinations`	Couples cluster + namespace autorisés (jokers acceptés)
`clusterResourceWhitelist`	Ressources de niveau cluster autorisées (vide = aucune)
`roles`	Rôles et jetons propres au projet

⚠ Piège courant

projet `default` = aucune restriction (à verrouiller en production)

🕮 Le projet `flotte`

Dépôt unique, namespaces `flotte-*`, création de namespace seulement

```yaml
apiVersion: argoproj.io/v1alpha1
kind: AppProject
metadata: {name: flotte, namespace: argocd}
spec:
  sourceRepos: ["https://gitlab.com/<GITLAB_USER>/flotte-config.git"]
  destinations:
    - {server: "https://kubernetes.default.svc", namespace: "flotte-*"}
  clusterResourceWhitelist:
    - {group: "", kind: Namespace}
  orphanedResources: {warn: true}
```

- RBAC (qui ?) + AppProject (où, quoi ?) = double barrière

🕮 Le projet en action

Refus explicites à la création de l'application

TENTATIVE DE `DEV-ALICE`	RÉSULTAT
Namespace `flotte-dev`, dépôt autorisé	Application créée
Namespace `default`	`do not match any of the allowed destinations`
Dépôt non déclaré	`application repo <URL> is not permitted`
Suppression de l'application	`PermissionDenied` (RBAC)

- Ordre des contrôles : RBAC (droit d'agir) → projet (cible conforme)

🕮 Déclarer un dépôt privé

Un Secret étiqueté, jamais de dépôt dans `argocd-cm` (supprimé en 3.0)

```yaml
apiVersion: v1
kind: Secret
metadata:
  name: repo-flotte-config
  namespace: argocd
  labels: {argocd.argoproj.io/secret-type: repository}
stringData:
  type: git
  url: https://gitlab.com/<GITLAB_USER>/flotte-config.git
  project: flotte
  username: <GITLAB_DEPLOY_USER>
  password: <GITLAB_DEPLOY_TOKEN>
```

- Jeton de déploiement GitLab, portée `read_repository` (moindre privilège)

🕮 Dépôts : vérifier la connexion

`STATUS` = test de connexion réel (`git ls-remote`)

```bash
argocd repo list
```

Sortie attendue (colonnes principales) :

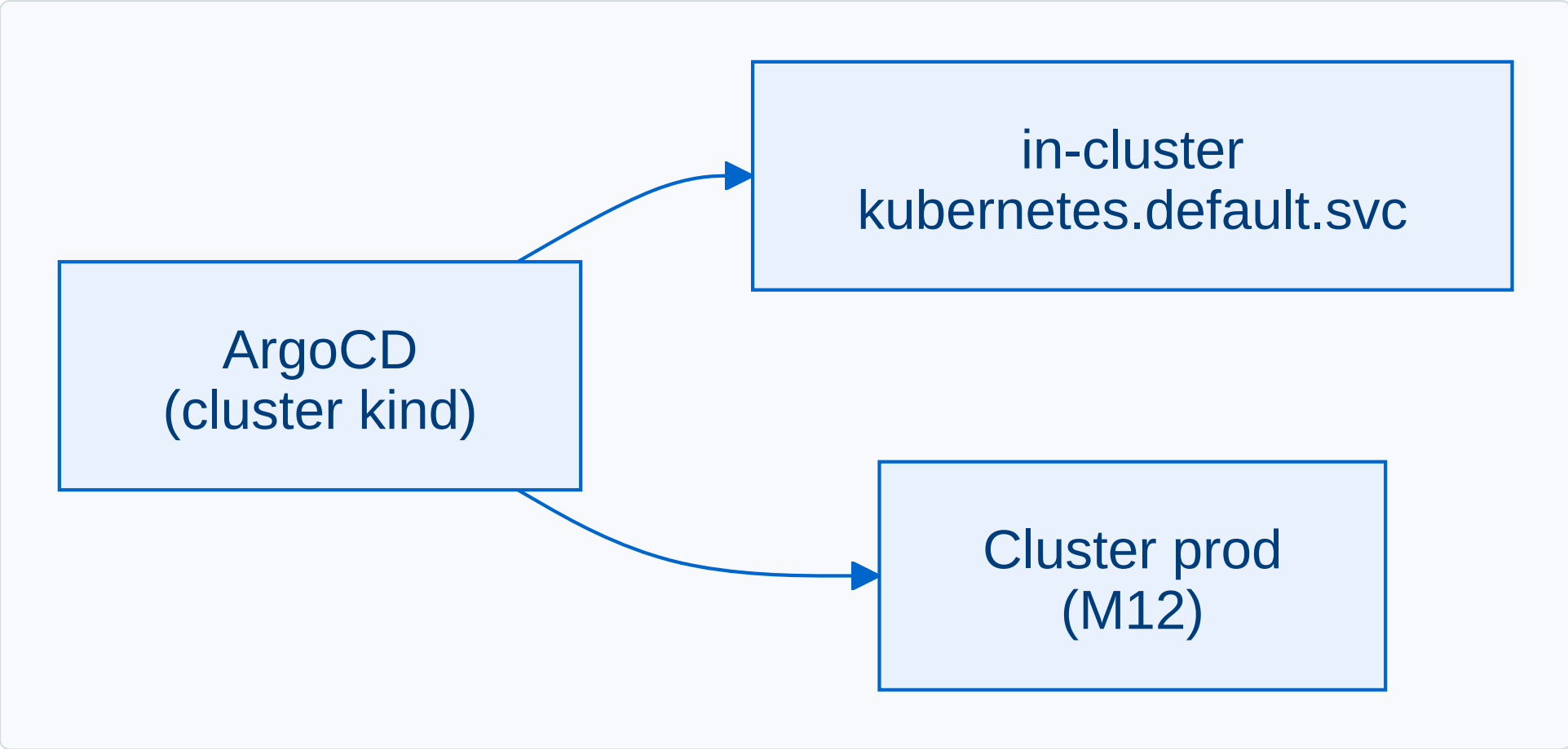
```
REPO                                                CREDS  STATUS      PROJECT
https://gitlab.com/<GITLAB_USER>/flotte-config.git  true   Successful  flotte
```

- Alternatives : `argocd repo add` (CLI), *Settings* → *Repositories* (UI)
- `repo-creds` : identifiants communs à tous les dépôts d'un préfixe d'URL

> **⚠ Piège courant**
>
> Secret créé par l'UI = hors Git (à recréer après reconstruction)

🕮 Les clusters gérés

Le cluster local est déclaré d'office, les autres par Secret ou CLI




```
argocd cluster list
```

À retenir

Points clés

- `argocd-cm` : patcher, jamais réécrire (valeurs par défaut en 3.x)
- Comptes : `login` pour les humains, `apiKey` pour la CI ; `admin` désactivé ensuite
- RBAC = `p` (permission d'un rôle) + `g` (attribution) ; `policy.default` vide
- AppProject = où et quoi ; RBAC = qui : double barrière
- Dépôts et clusters = Secrets étiquetés, versionnables

QCM d'ancrage (1/4)

Questions 1 à 2

1. Pourquoi patcher `argocd-cm` plutôt que l'appliquer en entier en 3.x ?

- **A.** `kubectl apply` est interdit sur les ConfigMaps
- **B.** Il contient des valeurs par défaut (exclusions) qui seraient supprimées
- **C.** Le patch redémarre ArgoCD automatiquement
- **D.** `argocd-cm` est en lecture seule

Réponse B — En 3.x, `argocd-cm` contient des exclusions et personnalisations par défaut ; un apply partiel les supprime.

Révéler la réponse

2. Quelle modification exige un redémarrage des pods concernés ?

- **A.** `ui.bannercontent` dans `argocd-cm`
- **B.** Une règle dans `argocd-rbac-cm`
- **C.** `server.insecure` dans `argocd-cmd-params-cm`
- **D.** Un nouveau Secret de dépôt

Réponse C — `argocd-cmd-params-cm` est lu au démarrage des composants : redémarrage nécessaire.

Révéler la réponse

QCM d'ancrage (2/4)

Questions 3 à 4

3. Quelle capacité donner au compte d'un pipeline CI ?

- **A.** `login`
- **B.** `apiKey`
- **C.** `admin`
- **D.** `sync`

Réponse B — `apiKey` permet de générer des jetons sans connexion interactive.

Révéler la réponse

4. Que fait la ligne `g, dev-alice, role:dev` ?

- **A.** Crée le compte `dev-alice`
- **B.** Attribue le rôle `role:dev` à `dev-alice`
- **C.** Donne tous les droits à `dev-alice`
- **D.** Supprime `dev-alice` du groupe `dev`

Réponse B — Une ligne `g` attribue un rôle (ou un groupe) à un sujet ; le compte est déclaré dans `argocd-cm`.

Révéler la réponse

QCM d'ancrage (3/4)

Questions 5 à 6

5. Quel est l'objet d'une règle sur `applications` ?

- **A.** Le nom du namespace
- **B.** `<projet>/<application>`
- **C.** L'URL du dépôt Git
- **D.** Le nom du cluster

Réponse B — Pour applications, applicationsets, logs et exec, l'objet est `<projet>/<application>`.

Révéler la réponse

6. `dev-alice` (droits sur `flotte/*`) cible le namespace `default`. Que se passe-t-il ?

- **A.** L'application est créée dans `default`
- **B.** Refus : destination non autorisée par le projet
- **C.** Refus : compte désactivé
- **D.** L'application est créée dans `flotte-default`

Réponse B — La destination `default` ne correspond pas au motif `flotte-*` du projet : `InvalidSpecError`.

Révéler la réponse

QCM d'ancrage (4/4)

Questions 7 à 8

7. Où déclarer un dépôt Git privé dans ArgoCD 3.x ?

A. Clé `repositories` de `argocd-cm`

B. Secret étiqueté `argocd.argoproj.io/secret-type: repository`

C. ConfigMap `argocd-rbac-cm`

D. Annotation de l'Application

Réponse B — Depuis 3.0, la déclaration des dépôts dans `argocd-cm` est supprimée ; seuls les Secrets étiquetés subsistent.

Révéler la réponse

8. Que permet le projet `default` livré à l'installation ?

A. Rien, il est verrouillé

B. Uniquement le namespace `default`

C. Tous les dépôts, toutes les destinations, toutes les ressources

D. Uniquement la lecture

Réponse C — Le projet `default` autorise tous les dépôts, destinations et ressources : à restreindre en production.

Révéler la réponse

🎓 Module 4 : Manifestes d'application

Durée

1 h 30 (S4.1 : 45 min · S4.2 : 45 min)

Prérequis

M03 terminé (projet `flotte`, dépôt `flotte-config` connecté)

Fil rouge

premier déploiement de `flotte-web` par ArgoCD

Objectifs

- Rédiger les manifestes Kubernetes de `flotte-web` et les pousser dans Git
- Écrire une ressource `Application` déclarative conforme au projet `flotte`
- Lire l'écart Git ↔ cluster avant de synchroniser (`app diff`)
- Reconnaître les formats de source : répertoire YAML, chart Helm, Kustomize

Déroulé

- Séquence S4.1 — La ressource Application (45 min)
- Séquence S4.2 — Formats de configuration (45 min)

Séquence S4.1

La ressource Application

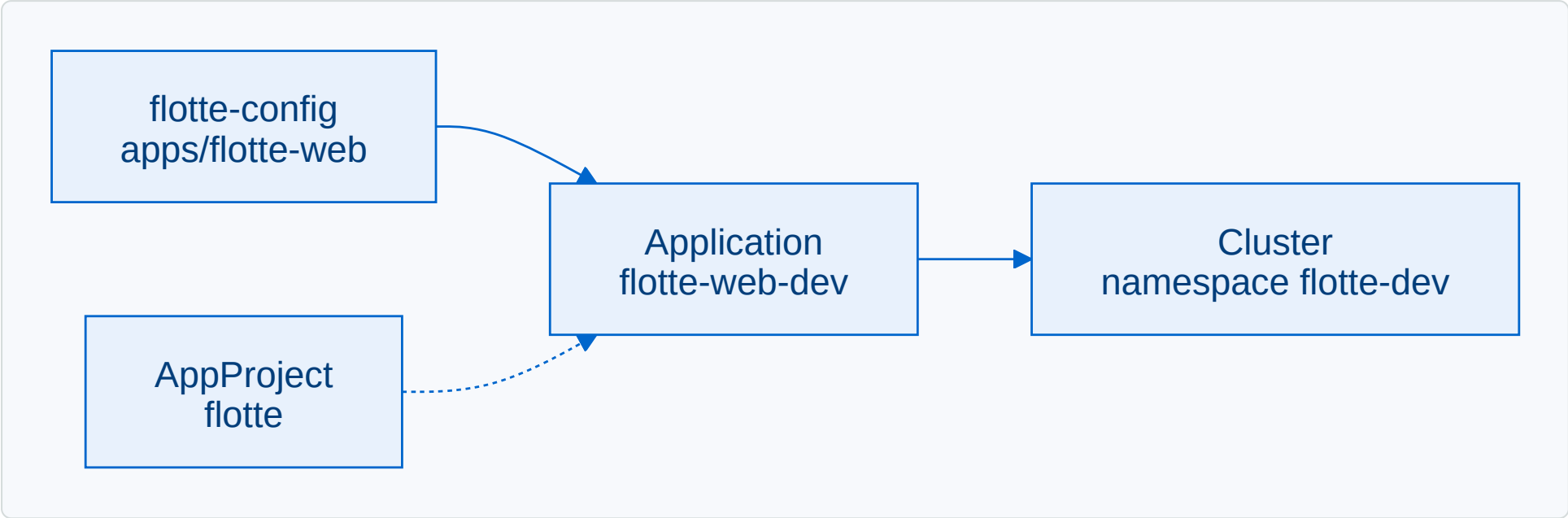
45 min

1. Qu'est-ce qu'une Application ?
2. Les manifestes de `flotte-web`
3. Anatomie d'une Application
4. Les champs clés
5. Quelle révision suivre ?
6. Trois façons de créer une Application
7. Avant de synchroniser : lire l'écart
8. Premier déploiement

❓ Qu'est-ce qu'une Application ?

Le lien entre un emplacement Git et un emplacement du cluster

```
Application = Source (dépôt + révision + chemin) + Destination (cluster + namespace) + Projet
```



- Ressource Kubernetes (CRD) dans le namespace `argocd`
- Une Application = une unité de synchronisation, de santé et d'historique

📖 Les manifestes de `flotte-web`

Trois fichiers dans `flotte-config/apps/flotte-web/`

FICHIER	RESSOURCE	RÔLE
`deployment.yaml`	Deployment	2 réplicas de `podinfo:6.15.0`, sondes, ressources
`service.yaml`	Service	Port 80 → port `http` (9898)
`configmap.yaml`	ConfigMap	Message et couleur de l'UI

```bash
kubectl apply --dry-run=server -n default -f apps/flotte-web/
```

Sortie attendue :

```
configmap/flotte-web-config created (server dry run)
deployment.apps/flotte-web created (server dry run)
service/flotte-web created (server dry run)
```

📖 Anatomie d'une Application

flotte-config/argocd/flotte-web-dev.yaml

```yaml
apiVersion: argoproj.io/v1alpha1
kind: Application
metadata:
  name: flotte-web-dev
  namespace: argocd
  finalizers: [resources-finalizer.argocd.argoproj.io]
spec:
  project: flotte
  source: {repoURL: "https://gitlab.com/<GITLAB_USER>/flotte-config.git",
           targetRevision: main, path: apps/flotte-web}
  destination: {server: "https://kubernetes.default.svc", namespace: flotte-dev}
  syncPolicy: {syncOptions: [CreateNamespace=true]}
```

📖 Les champs clés

Ce que vérifie ArgoCD à la création

CHAMP	EXEMPLE	CONTRÔLE
`spec.project`	`flotte`	RBAC + règles du projet
`source.repoURL`	URL de `flotte-config`	Dans `sourceRepos` du projet
`source.targetRevision`	`main`	Branche, tag ou commit existant
`source.path`	`apps/flotte-web`	Dossier existant dans la révision
`destination.server` / `name`	`https://kubernetes.default.svc`	Cluster déclaré
`destination.namespace`	`flotte-dev`	Motif `flotte-*` du projet

❓ Quelle révision suivre ?

`targetRevision` décide de ce qui est déployé

VALEUR	EXEMPLE	EFFET
Branche	`main`	Suit chaque nouveau commit
Tag	`v1.4.0`	Figé jusqu'au changement du tag suivi
Commit	`05b6219`	Figé, reproductible
`HEAD`	`HEAD`	Branche par défaut du dépôt

- Tag Git déplacé = déploiement modifié sans commit visible (tags à protéger)

💡 Bonne pratique

branche pour `dev`, tag ou commit pour `prod` (M09)

🕮 Trois façons de créer une Application

Même ressource, traçabilité différente

MÉTHODE	COMMANDE	TRACE DANS GIT
UI	*+ New App*	Non
CLI	`argocd app create <NOM> --repo <URL> --path <CHEMIN>`	Non
Déclarative	`kubectl apply -f argocd/flotte-web-dev.yaml`	Oui

- Déclaratif = Application versionnée, revue, reconstructible
- Étape suivante : Applications créées par ArgoCD lui-même (*App of Apps*, M12)

En entreprise

création UI réservée aux essais ; production = YAML dans Git

📖 Avant de synchroniser : lire l'écart

`OutOfSync` / `Missing` : rien n'existe encore dans le cluster

```bash
kubectl apply -f argocd/flotte-web-dev.yaml
argocd app get flotte-web-dev
argocd app diff flotte-web-dev
```

Sortie attendue (extrait de `app get`) :

```
Sync Status:        OutOfSync from main (ef64ceb)
Health Status:      Missing

GROUP  KIND        NAMESPACE   NAME               STATUS     HEALTH   HOOK  MESSAGE
       ConfigMap   flotte-dev  flotte-web-config  OutOfSync  Missing
       Service     flotte-dev  flotte-web         OutOfSync  Missing
apps   Deployment  flotte-dev  flotte-web         OutOfSync  Missing
```

∞ Premier déploiement

Synchronisation manuelle par l'équipe dev

```bash
argocd app sync flotte-web-dev
argocd app wait flotte-web-dev --health
kubectl get deploy -n flotte-dev
```

Sortie attendue (extrait) :

```
Sync Status:        Synced to main (ef64ceb)
Health Status:      Healthy
NAME         READY   UP-TO-DATE   AVAILABLE   AGE
flotte-web   2/2     2            2           20s
```

- Annotation posée sur chaque ressource : `argocd.argoproj.io/tracking-id` (suivi 3.x)
- Finalizer : `argocd app delete` supprime aussi les ressources (cascade)

Séquence S4.2

Formats de configuration

45 min

1. Quel format pour la source ?
2. Répertoire de manifestes
3. Chart Helm depuis un dépôt de charts
4. Kustomize
5. Voir ce qu'ArgoCD va appliquer
6. Des manifestes prêts pour GitOps

❓ Quel format pour la source ?

Détection automatique par le `repo-server`

CONTENU DU CHEMIN	TYPE DÉTECTÉ	MODULE
`Chart.yaml`	Helm	M09
`kustomization.yaml`	Kustomize	M09
Fichiers `.yaml`, `.yml`, `.json`, `.jsonnet`	Répertoire (*Directory*)	M04

- Aussi : chart d'un dépôt Helm ou OCI (champ `chart`), plugin de configuration



🕮 Répertoire de manifestes

Le cas le plus simple : YAML brut

```yaml
source:
  repoURL: https://gitlab.com/<GITLAB_USER>/flotte-config.git
  path: apps/flotte-web
  directory:
    recurse: true
    exclude: "{*.md,tests/*}"
```

- Sans `recurse` : sous-dossiers ignorés
- `include` / `exclude` : motifs glob, accolades pour plusieurs motifs

> **⚠ Piège courant**
>
> manifeste rangé dans un sous-dossier sans `recurse: true` = jamais déployé, sans erreur

📖 Chart Helm depuis un dépôt de charts

Aucune copie du chart : ArgoCD le télécharge

```yaml
source:
  repoURL: https://stefanprodan.github.io/podinfo
  chart: podinfo
  targetRevision: 6.16.0
  helm:
    releaseName: flotte-web
    valuesObject:
      replicaCount: 1
      ui: {message: "Hexa Mobilite - apercu Helm"}
```

- `targetRevision` = version du chart (pas une branche)
- ArgoCD rend le chart (`helm template`) : aucun `helm install` , `helm list` vide

📖 Kustomize

Base commune + variantes par environnement (détail au M09)

```yaml
source:
  repoURL: https://github.com/stefanprodan/podinfo.git
  targetRevision: 6.17.0
  path: kustomize
```

- `kustomization.yaml` présent dans `path` → Kustomize appliqué
- Options ArgoCD : `kustomize.images`, `namePrefix`, `commonLabels`

🔄 Avant / Après

Avant: une copie des manifestes par environnement (dérive entre copies)
Après: `base/` + `overlays/dev`, `overlays/prod` (écarts explicites)

📖 Voir ce qu'ArgoCD va appliquer

Le rendu final, quel que soit le format

```bash
argocd app manifests flotte-web-helm --source git | grep -E '^kind:|image:'
```

Sortie attendue :

```
kind: Service
kind: Deployment
        image: ghcr.io/stefanprodan/podinfo:6.16.0
```

- `--source git` : rendu depuis Git ; `--source live` : état du cluster
- Résumé des images déployées : `argocd app get <APP> -o yaml` (champ `status.summary.images`)

📖 Des manifestes prêts pour GitOps

Ce qui rend une application pilotable par ArgoCD

- Image avec tag immuable (`6.15.0`), jamais `latest` (aucun changement détectable dans Git)
- Sondes `readinessProbe` / `livenessProbe` : la santé ArgoCD en dépend
- `requests` / `limits` déclarés
- Labels communs (`app.kubernetes.io/name` , `part-of`)
- Aucun champ `namespace` en dur : fourni par la destination
- Aucun Secret en clair (M11)

À retenir

Points clés

- Application = source (dépôt, révision, chemin) + destination + projet
- YAML dans Git + `kubectl apply` : Applications déclaratives et reconstructibles
- `argocd app diff` avant toute synchronisation
- Format détecté par le contenu du chemin : répertoire, Helm, Kustomize
- `targetRevision` : branche (suivi), tag ou commit (figé)

QCM d'ancrage (1/4)

Questions 1 à 2

1. Dans quel namespace est créée la ressource `Application` ?

- **A.** Le namespace de destination (`flotte-dev`)
- **B.** `argocd`
- **C.** `default`
- **D.** `kube-system`

Réponse B — L'Application est une ressource ArgoCD du namespace `argocd` ; `flotte-dev` est sa destination.

Révéler la réponse

2. Une Application vient d'être créée sans synchronisation. Quels statuts affiche-t-elle ?

- **A.** `Synced` / `Healthy`
- **B.** `OutOfSync` / `Missing`
- **C.** `Unknown` / `Degraded`
- **D.** `Synced` / `Missing`

Réponse B — Ressources absentes du cluster : `OutOfSync` (écart avec Git) et `Missing` (santé).

Révéler la réponse

QCM d'ancrage (2/4)

Questions 3 à 4

3. Quelle valeur de `targetRevision` suit chaque nouveau commit ?

- **A.** Un hash de commit
- **B.** Un tag `v1.4.0`
- **C.** Une branche `main`
- **D.** Une version de chart `6.16.0`

Réponse C — Une branche est réévaluée à chaque réconciliation ; tag et commit figent la révision.

Révéler la réponse

4. Comment ArgoCD sait-il qu'un chemin contient un chart Helm ?

- **A.** Présence d'un fichier `Chart.yaml`
- **B.** Nom du dossier `helm/`
- **C.** Annotation sur l'Application
- **D.** Extension `.tgz` obligatoire

Réponse A — Le `repo-server` détecte Helm à la présence de `Chart.yaml`.

Révéler la réponse

QCM d'ancrage (3/4)

Questions 5 à 6

5. Que permet le finalizer `resources-finalizer.argocd.argoproj.io` ?

- **A.** Empêcher toute suppression de l'Application
- **B.** Supprimer les ressources déployées lors de la suppression de l'Application
- **C.** Synchroniser automatiquement
- **D.** Chiffrer les manifestes

Réponse B — Le finalizer déclenche la suppression en cascade des ressources gérées.

Révéler la réponse

6. Pourquoi éviter le tag d'image `latest` en GitOps ?

- **A.** Il est refusé par Kubernetes
- **B.** Une nouvelle image ne modifie pas Git : aucun écart détecté, aucune trace
- **C.** Il est plus lent à télécharger
- **D.** ArgoCD ne sait pas lire les tags

Réponse B — `latest` : nouvelle image sans modification de Git, donc ni écart détecté ni historique.

Révéler la réponse

QCM d'ancrage (4/4)

Questions 7 à 7

7. Quelle commande affiche les manifestes rendus depuis Git pour une Application ?

A. `argocd app manifests <APP> --source git`

B. `kubectl get <APP> -o yaml`

C. `helm list`

D. `argocd repo get <URL>`

Réponse A — `argocd app manifests <APP> --source git` affiche le rendu des manifestes depuis Git.

Révéler la réponse

🎓 Module 5 : Synchronisation

Durée
1 h 45 (S5.1 : 45 min · S5.2 : 1 h)

Prérequis
M04 terminé (`flotte-web-dev` `Synced` / `Healthy`)

Fil rouge
passer `flotte-web` en livraison continue

Objectifs

- Activer la synchronisation automatique et en expliquer les règles
- Corriger une dérive par self-heal et supprimer l'obsolète par prune
- Choisir les options de synchronisation adaptées à un contexte
- Ordonner un déploiement par hooks (PreSync, PostSync) et sync waves

Déroulé

- Séquence S5.1 — Manuelle, automatique, prune, self-heal (45 min)
- Séquence S5.2 — Options, hooks et sync waves (1 h)

Séquence S5.1

Manuelle, automatique, prune, self-heal

45 min

1. Manuelle ou automatique ?
2. Activer la synchronisation automatique
3. Les règles de l'automatisme
4. Dérive et self-heal
5. Prune : supprimer ce qui a quitté Git
6. Ignorer une différence voulue
7. Quelle politique pour quel environnement ?

❓ Manuelle ou automatique ?

Qui déclenche l'application de Git sur le cluster

`Sync automatique = Commit détecté → Application sans action humaine`

POLITIQUE	DÉCLENCHEUR	USAGE TYPE
Manuelle (défaut)	Bouton *Sync*, `argocd app sync`	Production sous contrôle, découverte
Automatique	Écart détecté (réconciliation, webhook)	Dev, recette, production GitOps mature
+ `prune`	Ressource retirée de Git → supprimée	Environnements sans ressource manuelle
+ `selfHeal`	Modification du cluster → annulée	Environnements verrouillés

🕮 Activer la synchronisation automatique

Dans l'Application, donc dans Git

```yaml
spec:
  syncPolicy:
    automated:
      enabled: true      # false = pause sans perdre le réglage
      prune: true
      selfHeal: true
    syncOptions:
      - CreateNamespace=true
```

```bash
kubectl apply -f argocd/flotte-web-dev.yaml
```

💡 Bonne pratique

`enabled: false` pour suspendre l'automatisme pendant un incident

📖 Les règles de l'automatisme

Ce que fait, et ne fait pas, la sync automatique

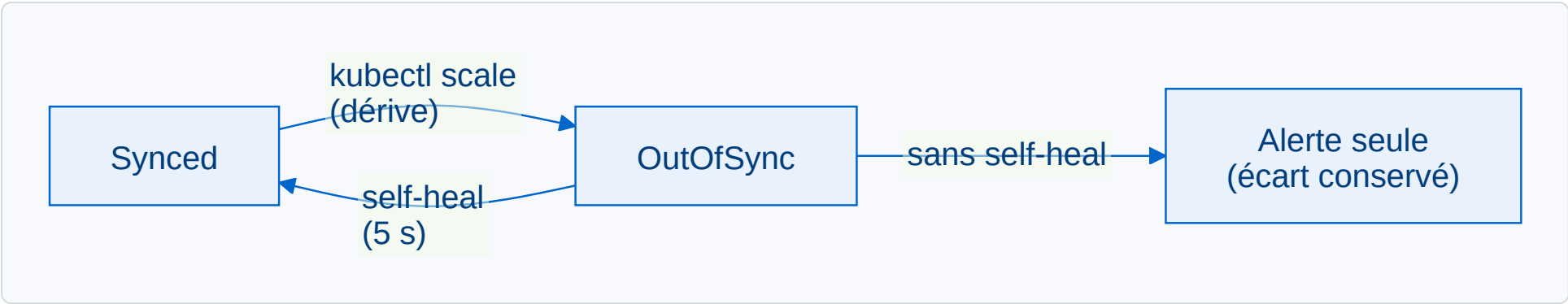
- Déclenchée uniquement si l'application est `OutOfSync`
- Une seule tentative par couple commit + paramètres (sauf `selfHeal`)
- Échec : 5 nouvelles tentatives par défaut, délai croissant (plusieurs minutes)
- Délai de détection : 120 s + jitter 60 s, ou immédiat avec webhook (M10)
- `argocd app rollback` refusé tant que l'automatisme est actif (M06)

⚠ Piège courant

sync automatique ≠ instantanée (attendre ou *Refresh*)

🕮 Dérive et self-heal

Le cluster s'écarte de Git, ArgoCD le ramène



Après `kubectl scale --replicas=5`, sortie de `argocd app diff` (sans self-heal) :

```
<   replicas: 5
---
>   replicas: 3
```

Prune : supprimer ce qui a quitté Git

Désactivé par défaut, par prudence

Avant / Après

Avant (sans prune): fichier retiré de Git → ressource conservée, `OutOfSync`
Après (avec prune): fichier retiré de Git → ressource supprimée, `Pruned`

```
GROUP  KIND       NAMESPACE   NAME               STATUS     HEALTH  HOOK  MESSAGE
       ConfigMap  flotte-dev  flotte-web-legacy  Succeeded  Pruned        pruned
```

- Effet à la réconciliation suivante après activation (ou *Refresh*)
- Protection d'une ressource : annotation `argocd.argoproj.io/sync-options: Prune=false`

Piège courant

prune + mauvais chemin `path` = suppression de toute l'application

🕮 Ignorer une différence voulue

Un autre acteur gère un champ (HPA, opérateur, mutation)

```yaml
spec:
  ignoreDifferences:
    - group: apps
      kind: Deployment
      name: flotte-web
      jsonPointers: [/spec/replicas]
  syncPolicy:
    syncOptions: [RespectIgnoreDifferences=true]
```

- Sans `RespectIgnoreDifferences` : écart ignoré dans le diff, mais écrasé à la prochaine sync
- Testé : `kubectl scale --replicas=4` → application `Synced`, 4 réplicas conservés

❓ Quelle politique pour quel environnement ?

Plus l'environnement est critique, plus la décision est explicite

ENVIRONNEMENT	POLITIQUE	RAISON
`dev`	Auto + prune + self-heal	Retour rapide, aucune modification manuelle
`recette`	Auto + prune	Correctifs manuels tolérés le temps d'un test
`prod`	Manuelle ou auto + fenêtres de sync	Livraison décidée, traçable (M12)

En entreprise

promotion en prod par merge request, la sync suit le merge

Séquence S5.2

Options, hooks et sync waves

1 h

1. Options de synchronisation
2. Les phases d'une synchronisation
3. Déclarer un hook
4. Cycle de vie des hooks
5. Sync waves : ordonner dans une phase
6. Une synchronisation avec hooks
7. Hook en échec : livraison bloquée

🕮 Options de synchronisation

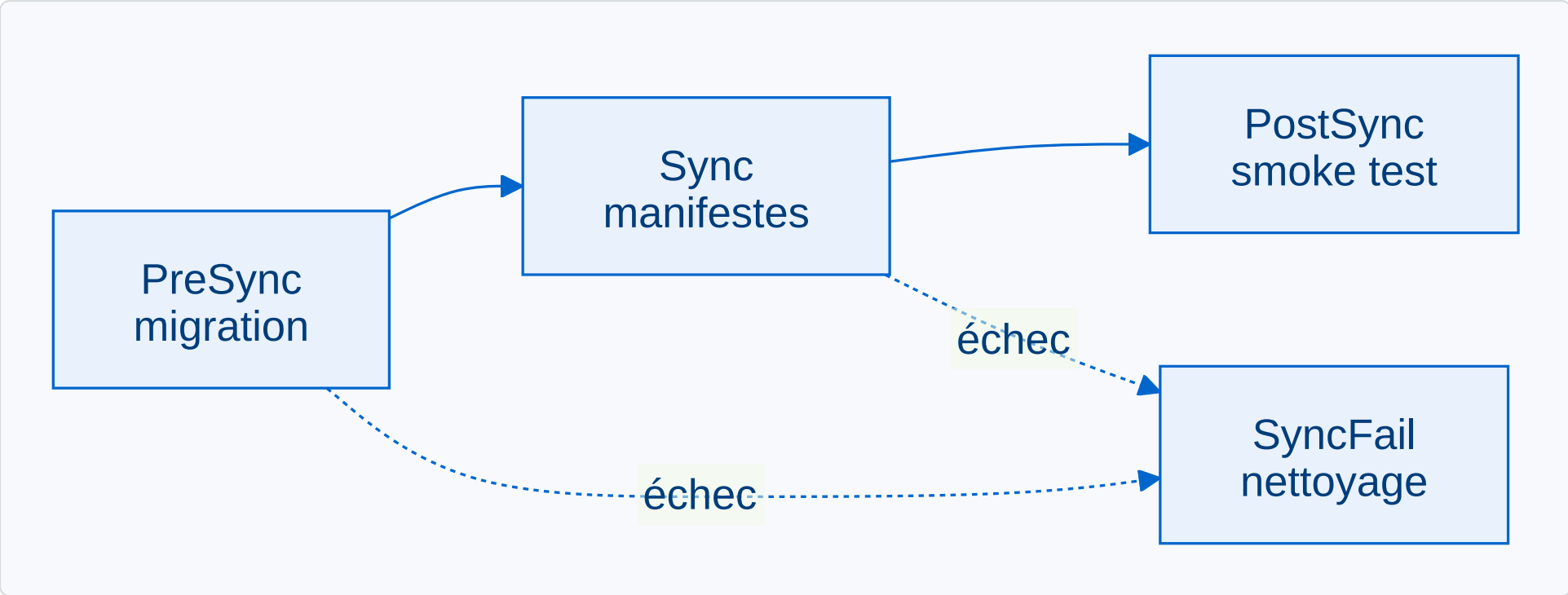
Au niveau de l'Application (`syncOptions`) ou d'une ressource (annotation)

OPTION	EFFET
`CreateNamespace=true`	Crée le namespace de destination
`ServerSideApply=true`	`kubectl apply --server-side` (gros objets, CRD)
`ApplyOutOfSyncOnly=true`	N'applique que les ressources en écart (grandes applications)
`PruneLast=true`	Supprime après le déploiement des nouvelles ressources
`Replace=true`	`kubectl replace` / `create` au lieu d' `apply`

- Annotation ressource : `argocd.argoproj.io/sync-options: Prune=false,Delete=false`

🕮 Les phases d'une synchronisation

Avant, pendant, après, en cas d'échec



- PostSync : après application ET ressources `Healthy`
- `PreDelete` / `PostDelete` : à la suppression de l'Application uniquement
- Hooks non exécutés lors d'une synchronisation sélective (ressources choisies)

🕮 Déclarer un hook

Une ressource (souvent un Job) + une annotation

```yaml
apiVersion: batch/v1
kind: Job
metadata:
  name: flotte-db-migration
  annotations:
    argocd.argoproj.io/hook: PreSync
    argocd.argoproj.io/hook-delete-policy: BeforeHookCreation
spec:
  template:
    spec:
      restartPolicy: Never
      containers: [{name: migration, image: busybox:1.37, command: [sh, -c, "echo OK"]}]
```

🕮 Cycle de vie des hooks

Quand le hook est-il supprimé ?

POLITIQUE	SUPPRESSION
`BeforeHookCreation` (défaut)	Avant la création du hook suivant (dernier conservé)
`HookSucceeded`	Dès la réussite
`HookFailed`	Après un échec

- Nom fixe + `BeforeHookCreation` : dernier résultat consultable (`kubectl logs job/<NOM_DU_JOB>`)
- `generateName` : un nouveau Job à chaque sync (prévoir une politique de suppression)

⚠ Piège courant

`ttlSecondsAfterFinished` sur un hook = Job supprimé avant lecture du résultat

🕮 Sync waves : ordonner dans une phase

Vague la plus basse d'abord, 2 s entre deux vagues

Ordre = Phase → Vague (sync-wave) → Type de ressource → Nom




```yaml
metadata:
  annotations:
    argocd.argoproj.io/sync-wave: "-1"
```

- Vague suivante seulement si la précédente est `Healthy`
- Prune : ordre inverse (vagues hautes supprimées d'abord)

🕮 Une synchronisation avec hooks

Résultat testé sur `flotte-web-dev`

```
KIND        NAME                 STATUS     HEALTH   HOOK
Job         flotte-db-migration  Succeeded  Synced   PreSync
ConfigMap   flotte-web-config    Synced
Service     flotte-web           Synced     Healthy
Deployment  flotte-web           Synced     Healthy
Job         flotte-smoke-test    Succeeded  Synced   PostSync
```

- Journal du hook : `kubectl logs -n flotte-dev job/flotte-db-migration`
- UI : panneau *Sync status* → détail des phases et des vagues

🕮 Hook en échec : livraison bloquée

Le garde-fou d'une migration ratée

⇄ Avant / Après

Avant: image `6.16.0` poussée + migration en échec → Deployment toujours en `6.15.0`
Après: migration corrigée + nouveau commit → `6.16.0` déployée, smoke test réussi

```
Running | one or more synchronization tasks completed unsuccessfully.
          Retrying attempt #5 at 2026-11-08T12:52:49Z.
```

- Correction = nouveau commit (même commit en échec non retenté automatiquement)
- Arrêt des tentatives en cours : `argocd app terminate-op flotte-web-dev`

À retenir

Points clés

- Auto-sync = une tentative par commit, 5 nouvelles tentatives en cas d'échec
- Self-heal annule les dérives ; prune supprime ce qui a quitté Git
- `ignoreDifferences` + `RespectIgnoreDifferences=true` pour un champ géré ailleurs
- Hooks : PreSync (préparer), PostSync (vérifier), SyncFail (nettoyer)
- Sync waves : ordre au sein d'une phase, vague suivante si la précédente est saine

QCM d'ancrage (1/4)

Questions 1 à 2

1. Une ressource est supprimée de Git, `prune` est désactivé. Que se passe-t-il ?

- **A.** La ressource est supprimée du cluster
- **B.** La ressource reste, l'application passe `OutOfSync`
- **C.** La synchronisation échoue
- **D.** ArgoCD recrée le fichier dans Git

Réponse B — Sans prune, la ressource orpheline reste en place et l'application est `OutOfSync`.

Révéler la réponse

2. Que fait `selfHeal` après un `kubectl scale` manuel ?

- **A.** Rien, il ne surveille que Git
- **B.** Il remet la valeur de Git (quelques secondes)
- **C.** Il modifie Git pour refléter le cluster
- **D.** Il supprime le Deployment

Réponse B — Self-heal resynchronise environ 5 s après la détection de l'écart.

Révéler la réponse

QCM d'ancrage (2/4)

Questions 3 à 4

3. Quel champ suspend la synchronisation automatique sans perdre son réglage ?

- **A.** `automated.enabled: false`
- **B.** `selfHeal: false`
- **C.** `prune: false`
- **D.** `syncOptions: [Manual=true]`

Réponse A — `enabled: false` suspend l'automatisme en conservant `prune` et `selfHeal`.

Révéler la réponse

4. Dans quelle phase placer une migration de base de données ?

- **A.** `PostSync`
- **B.** `PreSync`
- **C.** `SyncFail`
- **D.** `PostDelete`

Réponse B — `PreSync` s'exécute avant l'application des manifestes.

Révéler la réponse

QCM d'ancrage (3/4)

Questions 5 à 6

5. Quel est l'effet d'un hook `PreSync` en échec ?

- **A.** Les manifestes sont appliqués quand même
- **B.** La synchronisation s'arrête, les ressources ne sont pas mises à jour
- **C.** Le hook est ignoré
- **D.** L'application est supprimée

Réponse B — Un hook `PreSync` en échec arrête la synchronisation : les ressources ne sont pas modifiées.

Révéler la réponse

6. Quelle vague est appliquée en premier ?

- **A.** `"5"`
- **B.** `"0"`
- **C.** `"-1"`
- **D.** L'ordre alphabétique des fichiers

Réponse C — Les vagues sont appliquées de la plus basse à la plus haute.

Révéler la réponse

QCM d'ancrage (4/4)

Questions 7 à 8

7. Pourquoi ajouter `RespectIgnoreDifferences=true` avec `ignoreDifferences` ?

- **A.** Pour masquer le champ dans l'UI
- **B.** Pour que la synchronisation n'écrase pas le champ ignoré
- **C.** Pour activer le prune
- **D.** Pour accélérer la réconciliation

Réponse B — Sans cette option, l'écart est seulement masqué dans le diff, puis écrasé lors d'une synchronisation.

Révéler la réponse

8. Quelle politique de suppression de hook s'applique si aucune n'est indiquée ?

- **A.** `HookSucceeded`
- **B.** `HookFailed`
- **C.** `BeforeHookCreation`
- **D.** Aucune suppression

Réponse C — Sans politique explicite, `BeforeHookCreation` s'applique.

Révéler la réponse

🎓 Module 6 : Versions et rollback

Durée

1 h 15 (S6.1 : 1 h 15)

Prérequis

M05 terminé (auto-sync, self-heal)

Fil rouge

la version `6.17.0` de `flotte-web` part en production avec une anomalie

Objectifs

- Lire l'historique des déploiements et comparer deux révisions
- Revenir à une version antérieure par `git revert` (voie GitOps)
- Effectuer un rollback d'urgence avec `argocd app rollback`, puis réaligner Git
- Restreindre le droit de rollback par le RBAC

Déroulé

- Séquence S6.1 — Historique, révisions et rollback (1 h 15)

Séquence S6.1

Historique, révisions et rollback

1 h 15

1. L'historique d'une application
2. Comparer deux révisions
3. Deux façons de revenir en arrière
4. Rollback GitOps : `git revert`
5. Rollback d'urgence : `argocd app rollback`
6. Après un rollback ArgoCD
7. Qui peut revenir en arrière ?
8. Versionner pour pouvoir revenir

📖 L'historique d'une application

Chaque synchronisation réussie = une entrée

Entrée d'historique = ID + date + révision Git (+ source)

```bash
argocd app history flotte-web-dev
```

Sortie attendue (extrait) :

```
ID      DATE                           REVISION
8       2026-11-08 13:58:58 +0100 CET  main (c0dcb75)
9       2026-11-08 14:01:13 +0100 CET  main (4ee1140)
10      2026-11-08 14:01:49 +0100 CET  main (3750a56)
```

- 10 entrées conservées par défaut (`spec.revisionHistoryLimit`)
- Nouvelle limite appliquée à la synchronisation suivante
- UI : bouton *History and rollback* de l'application

📖 Comparer deux révisions

Ce qu'une révision changerait par rapport au cluster

```bash
argocd app manifests flotte-web-dev --revision 4ee1140 | grep -E 'image: ghcr|MESSAGE'
argocd app diff flotte-web-dev --revision 4ee1140
```

Sortie attendue (extrait du diff) :

```
<   PODINFO_UI_MESSAGE: Hexa Mobilite - suivi de flotte
>   PODINFO_UI_MESSAGE: Hexa Mobilite - v6.17 (anomalie)
<         image: ghcr.io/stefanprodan/podinfo:6.16.0
>         image: ghcr.io/stefanprodan/podinfo:6.17.0
```

- Exemple : cluster en `6.16.0`, révision `4ee1140` (`6.17.0`)
- `<` cluster actuel, `>` révision demandée

📖 Deux façons de revenir en arrière

Même résultat immédiat, conséquences différentes

CRITÈRE	`GIT REVERT`	`ARGOCD APP ROLLBACK`
Source de vérité	Git reste la référence	Cluster ≠ Git (temporaire)
Auto-sync	Compatible	Refusé si actif
Trace	Commit visible, revu	Entrée d'historique ArgoCD
Délai	Réconciliation ou webhook	Immédiat
Usage	Cas nominal	Urgence, Git indisponible

💡 Bonne pratique

`git revert` par défaut ; `app rollback` = geste d'urgence suivi d'un revert

Rollback GitOps : `git revert`

Un commit qui annule un commit




```bash
git revert --no-edit HEAD && git push
argocd app get flotte-web-dev --refresh > /dev/null
```

Sortie attendue (`git log --oneline -2`) :

```
3750a56 Revert "feat: flotte-web 6.17.0"
4ee1140 feat: flotte-web 6.17.0
```

🕮 Rollback d'urgence : `argocd app rollback`

Redéployer une entrée de l'historique, sans passer par Git

```bash
argocd app rollback flotte-web-dev 8
```

Sortie attendue (auto-sync actif) :

```
rpc error: code = FailedPrecondition
  desc = rollback cannot be initiated when auto-sync is enabled
```

1. Suspendre l'automatisme (`automated.enabled: false`)
2. `argocd app rollback flotte-web-dev <ID>`
3. Corriger Git (`git revert`), puis réactiver l'automatisme

> **⚠ Piège courant**
>
> auto-sync réactivé AVANT le revert = retour de la version fautive

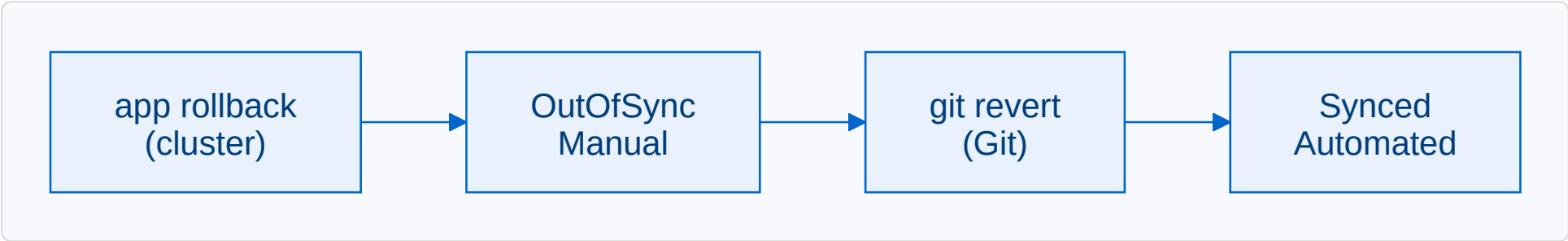
🕮 Après un rollback ArgoCD

Le cluster ne correspond plus à Git : c'est voulu, et temporaire

⇄ Avant / Après

Avant revert: `Sync Policy: Manual`, `OutOfSync from main` (Git = version fautive)

Après revert + réactivation: `Automated (Prune)`, `Synced to main` (Git = version saine)



❓ Qui peut revenir en arrière ?

Par défaut, `rollback` = permission `sync`

RÉGLAGE `argocd-cm`	CONTRÔLE APPLIQUÉ	`dev-alice` (SYNC SEUL)
Défaut	`sync`	Rollback autorisé
`server.rbac.rollback.enforce.enable: "true"`	`rollback`	`permission denied`

```
p, role:ops-release, applications, rollback, flotte/*, allow
```

- Option apparue récemment : vérifier la version avant de l'activer (testée en 3.6.1)
- `role:admin` intégré : inclut déjà `rollback`

📖 Versionner pour pouvoir revenir

Un retour arrière n'est simple que si la livraison est propre

- Une modification = un commit au message explicite (`feat: flotte-web 6.17.0`)
- Image et configuration dans le MÊME commit : un revert annule les deux
- Tags d'image immuables : `6.16.0` désigne toujours le même contenu
- Tags Git de release (`v1.4.0`) pour la production (M09)
- ConfigMap modifiée seule (`envFrom`) : pods NON redémarrés (générateur Kustomize, M09)

En entreprise

rollback testé en recette avant d'en avoir besoin en production

À retenir

Points clés

- `argocd app history` : ID, date, révision ; 10 entrées par défaut
- `app diff --revision` : comparer avant de revenir en arrière
- Voie normale : `git revert` + synchronisation automatique
- Urgence : `automated.enabled: false` → `app rollback <ID>` → `git revert` → réactivation
- Droit de rollback séparé : `server.rbac.rollback.enforce.enable: "true"`

QCM d'ancrage (1/4)

Questions 1 à 2

1. Pourquoi `argocd app rollback` est-il refusé sur `flotte-web-dev` en fin de M05 ?

- **A.** Le compte `admin` est désactivé
- **B.** La synchronisation automatique est active
- **C.** L'historique est vide
- **D.** Le dépôt Git est indisponible

Réponse B — `rollback cannot be initiated when auto-sync is enabled`.

Révéler la réponse

2. Quelle méthode garde Git comme source de vérité ?

- **A.** `argocd app rollback`
- **B.** `kubectl rollout undo`
- **C.** `git revert` puis synchronisation
- **D.** Modifier l'image avec `kubectl set image`

Réponse C — Le revert est un commit : Git reste la source de vérité.

Révéler la réponse

QCM d'ancrage (2/4)

Questions 3 à 4

3. Après `argocd app rollback`, quel est l'état de l'application ?

- **A.** `Synced`, synchronisation automatique
- **B.** `OutOfSync`, Git contient toujours la version fautive
- **C.** `Unknown`
- **D.** Supprimée puis recréée

Réponse B — Le cluster exécute une ancienne révision, Git la version fautive : `OutOfSync`.

Révéler la réponse

4. Combien d'entrées d'historique sont conservées par défaut ?

- **A.** 3
- **B.** 5
- **C.** 10
- **D.** Illimité

Réponse C — `revisionHistoryLimit` vaut 10 par défaut.

Révéler la réponse

QCM d'ancrage (3/4)

Questions 5 à 6

5. Quelle commande affiche l'écart entre le cluster et une révision donnée ?

- **A.** `argocd app diff <APP> --revision <SHA>`
- **B.** `argocd app history <APP> --diff`
- **C.** `git diff <SHA>`
- **D.** `argocd app get <APP> --hard-refresh`

Réponse A — `argocd app diff --revision` compare le cluster à une révision Git donnée.

Révéler la réponse

6. Par défaut, quelle permission RBAC contrôle un rollback ?

- **A.** `update`
- **B.** `rollback`
- **C.** `sync`
- **D.** `override`

Réponse C — Sans `server.rbac.rollback.enforce.enable`, le rollback est contrôlé par `sync`.

Révéler la réponse

QCM d'ancrage (4/4)

Questions 7 à 7

7. Pourquoi réactiver la synchronisation automatique APRÈS le `git revert` ?

A. Pour accélérer la synchronisation

B. Sinon ArgoCD redéploie la version fautive encore présente dans Git

C. Pour vider l'historique

D. Parce que le revert exige un mode manuel

Réponse B — L'automatisme resynchronise immédiatement sur la version encore présente dans Git.

Révéler la réponse

🎓 Module 7 : Supervision et dépannage

Durée

1 h 45 (S7.1 : 1 h · S7.2 : 45 min)

Prérequis

M06 terminé (`flotte-web-dev` en `6.16.0` , auto-sync)

Fil rouge

Hexa Mobilité veut savoir AVANT ses clients quand `flotte-web` va mal

Objectifs

- Interpréter les statuts Sync et Health et la santé de chaque ressource
- Diagnostiquer une application `Degraded` ou en échec de synchronisation
- Configurer une notification webhook sur déploiement et sur dégradation
- Lire les métriques Prometheus d'ArgoCD et écrire une requête d'alerte

Déroulé

- Séquence S7.1 — Santé, diff, événements, journaux (1 h)
- Séquence S7.2 — Notifications et métriques (45 min)

Séquence S7.1

Santé, diff, événements, journaux

1 h

1. Lire l'état d'une application
2. Comment ArgoCD calcule la santé ?
3. Méthode de diagnostic
4. Où regarder ?
5. Panne 1 : image introuvable
6. Panne 2 : manifeste refusé
7. Symptômes et causes fréquentes

📖 Lire l'état d'une application

Deux axes indépendants, quatre situations

SYNC	HEALTH	SITUATION	PREMIER RÉFLEXE
`Synced`	`Healthy`	Nominal	—
`OutOfSync`	`Healthy`	Git ≠ cluster, service rendu	`argocd app diff`
`Synced`	`Degraded`	Git appliqué, application en panne	Événements, journaux
`OutOfSync`	`Degraded` / `Missing`	Synchronisation en échec ou en cours	Message d'opération

- `Progressing` : changement en cours (pods en démarrage, rollout)

❓ Comment ArgoCD calcule la santé ?

Une règle par type de ressource, l'application prend le pire statut

RESSOURCE	`HEALTHY` SI	`DEGRADED` SI
Deployment	Réplicas à jour et disponibles	`progressDeadlineSeconds` dépassé
Service `LoadBalancer`	IP ou nom attribué	—
Job (hook)	Terminé avec succès	Échec (`backoffLimit` atteint)
PersistentVolumeClaim	`Bound`	`Lost`

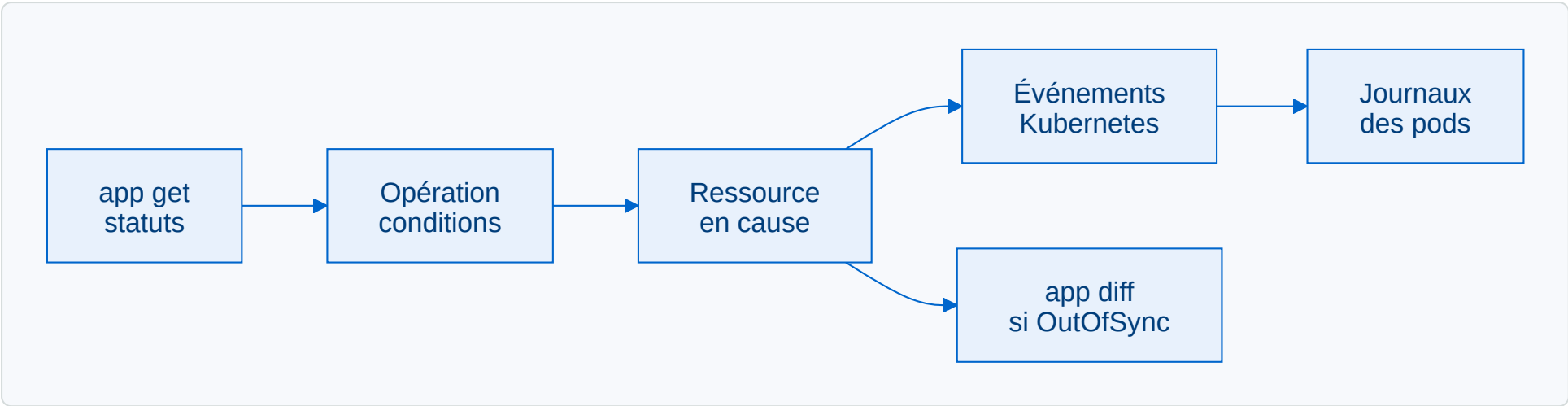
- Ressources personnalisées : règles Lua (`resource.customizations.health.<groupe>_<type>`)
- ConfigMap, Secret : pas de santé calculée

💡 Bonne pratique

`progressDeadlineSeconds` court en `dev` (60 s) = panne visible vite

🕮 Méthode de diagnostic

Du général au particulier



- UI : clic sur la ressource rouge → onglets *Summary*, *Events*, *Logs*, *Diff*

❓ Où regarder ?

Une commande par question

QUESTION	COMMANDE
Quel état, quelle opération ?	`argocd app get flotte-web-dev`
Quelles ressources gérées, orphelines ?	`argocd app resources flotte-web-dev`
Quel écart avec Git ?	`argocd app diff flotte-web-dev`
Que dit Kubernetes ?	`kubectl get events -n flotte-dev --sort-by=.lastTimestamp`
Que disent les pods ?	`argocd app logs flotte-web-dev --kind Deployment --name flotte-web`

- `argocd app logs` : soumis au RBAC `logs` (3.x) ; `audit-claire` et `dev-alice` autorisées

Panne 1 : image introuvable

`Synced` mais `Degraded` : Git est appliqué, l'image n'existe pas

```
Sync Status:        Synced to main (fe0af9e)
Health Status:      Degraded
apps   Deployment  flotte-dev  flotte-web           Synced   Degraded
Deployment "flotte-web" exceeded its progress deadline
```

```
Warning   Failed   pod/flotte-web-7d6589d6d8-jkb58
  Failed to pull image "ghcr.io/stefanprodan/podinfo:6.99.0": rpc error: code = NotFound
```

- Mise à jour progressive : anciens pods conservés, service toujours rendu
- `Progressing` jusqu'à l'échéance, puis `Degraded`

🕮 Panne 2 : manifeste refusé

`OutOfSync` : la synchronisation échoue à chaque tentative

```
Running | one or more synchronization tasks completed unsuccessfully,
reason: error when patching "/dev/shm/3761042127": Deployment.apps "flotte-web"
is invalid: spec.template.spec.containers: Required value.
Retrying attempt #4 at 2026-11-08T13:13:08Z.
```

- Cause : `containers:` mal orthographié (`containerz:`), champ inconnu ignoré, champ obligatoire absent
- Message lisible dans l'UI : bandeau *Sync failed → More*

⚠ Piège courant

champ inconnu = ignoré sans erreur ; seule l'absence d'un champ requis se voit

📖 Symptômes et causes fréquentes

Le réflexe : lire le MESSAGE avant d'agir

SYMPTÔME	CAUSE PROBABLE
`ComparisonError` : `authentication required`	Jeton du dépôt expiré
`ComparisonError` : `app path does not exist`	Chemin renommé dans Git
`OutOfSync` permanent sans changement Git	Champ modifié par un contrôleur (`ignoreDifferences`)
`Progressing` sans fin	Sondes en échec, `progressDeadlineSeconds` long
Opération `Running` bloquée	Hook en attente ou en échec (`terminate-op`)

Séquence S7.2

Notifications et métriques

45 min

1. Le moteur de notifications
2. Service et template
3. Triggers et abonnements
4. Le catalogue officiel
5. Métriques Prometheus
6. Requêtes utiles

🕮 Le moteur de notifications

Quand quelque chose se produit, prévenir quelqu'un

```
Notification = Trigger (quand) + Template (quoi) + Service (où) + Abonnement (qui)
```



- Configuration : ConfigMap `argocd-notifications-cm`, secrets dans `argocd-notifications-secret`

🕮 Service et template

Où envoyer, quoi envoyer

```yaml
service.webhook.flotte-alertes: |
  url: http://alertes-receiver.flotte-ops.svc.cluster.local/argocd
  headers: [{name: Content-Type, value: application/json}]
template.flotte-webhook: |
  webhook:
    flotte-alertes:
      method: POST
      body: |
        {"application": "{{.app.metadata.name}}",
         "health": "{{.app.status.health.status}}"}
```

- Variables : `{{.app.metadata.name}}`, `{{.app.status.sync.revision}}`

🕮 Triggers et abonnements

Quand déclencher, quelle application s'abonne

```yaml
trigger.flotte-degraded: |
  - when: app.status.health.status == 'Degraded'
    send: [flotte-webhook]
```

```yaml
metadata:
  annotations:
    notifications.argoproj.io/subscribe.flotte-degraded.flotte-alertes: ""
```

- Format : `subscribe.<trigger>.<service>: "<destinataire>"`
- `oncePer: app.status.sync.revision` : une notification par révision
- Abonnement dans le fichier de l'Application (Git) ou global (`subscriptions`)

📖 Le catalogue officiel

Triggers et templates prêts à l'emploi

```bash
CAT=https://raw.githubusercontent.com/argoproj/argo-cd/v3.6.1/notifications_catalog
kubectl apply -n argocd -f $CAT/install.yaml
```

- Triggers : `on-created`, `on-deleted`, `on-deployed`, `on-health-degraded`, `on-sync-failed`, `on-sync-running`, `on-sync-status-unknown`, `on-sync-succeeded`
- Services : Slack, Microsoft Teams, e-mail, Mattermost, webhook, GitHub, GitLab
- Remplace `argocd-notifications-cm` : l'appliquer AVANT ses propres clés

⚠ Piège courant

trigger maison nommé comme un trigger du catalogue = écrasé

🕮 Métriques Prometheus

Chaque composant expose `/metrics`

SERVICE	PORT	MÉTRIQUES CLÉS
`argocd-metrics` (contrôleur)	8082	`argocd_app_info`, `argocd_app_sync_total`
`argocd-server-metrics`	8083	Requêtes API et gRPC
`argocd-repo-server`	8084	Durée de rendu, requêtes Git

```
argocd_app_info{name="flotte-web-dev",project="flotte",health_status="Healthy",
  sync_status="Synced",autosync_enabled="true",phase="Succeeded"} 1
```

- 3.0 : `argocd_app_sync_status` et `argocd_app_health_status` supprimées → `argocd_app_info`

🕮 Requêtes utiles

De la métrique à l'alerte

BESOIN	REQUÊTE PROMQL
Applications en panne	`argocd_app_info{health_status="Degraded"}`
Applications en écart	`argocd_app_info{sync_status="OutOfSync"}`
Échecs de sync (1 h)	`sum by (name) (increase(argocd_app_sync_total{phase="Failed"}[1h]))`
Clusters injoignables	`argocd_cluster_connection_status == 0`

- Tableau de bord Grafana officiel : `examples/dashboard.json` du dépôt Argo CD
- Alerte = condition durable (`for: 10m`), pas un état transitoire

À retenir

Points clés

- Sync et Health se lisent ensemble ; la santé suit des règles par type de ressource
- Diagnostic : `app get` → message d'opération → ressource → événements → journaux
- Champ inconnu ignoré ; champ requis absent = synchronisation en échec
- Notifications = trigger + template + service + abonnement (annotation)
- Supervision Prometheus : `argocd_app_info` (3.x), `argocd_app_sync_total`

QCM d'ancrage (1/4)

Questions 1 à 2

1. Une application est `Synced` et `Degraded`. Où chercher en priorité ?

A. Dans `argocd app diff`

B. Dans les événements et journaux des pods

C. Dans la configuration RBAC

D. Dans l'historique Git

Réponse B — `Synced` : Git est appliqué ; la panne est à l'exécution (événements, journaux).

Révéler la réponse

2. Quand un Deployment passe-t-il `Degraded` pour ArgoCD ?

A. Dès qu'un pod redémarre

B. Quand `progressDeadlineSeconds` est dépassé sans progression

C. Quand il a plus de 3 réplicas

D. Quand la ConfigMap associée change

Réponse B — Le Deployment dépasse `progressDeadlineSeconds` sans progresser : `Degraded`.

Révéler la réponse

QCM d'ancrage (2/4)

Questions 3 à 4

3. `containers:` est écrit `containerz:` dans un Deployment. Quel est l'effet ?

A. Le champ inconnu est appliqué tel quel

B. Le champ inconnu est ignoré, la synchronisation échoue (containers requis)

C. ArgoCD corrige la faute automatiquement

D. L'application est supprimée

Réponse B — Le champ inconnu est ignoré ; l'absence de `containers` rend le manifeste invalide.

Révéler la réponse

4. Quelle commande lit les journaux des pods d'une application via ArgoCD ?

A. `argocd app logs <APP>`

B. `argocd app history <APP>`

C. `argocd app resources <APP>`

D. `argocd repo logs`

Réponse A — `argocd app logs` lit les journaux via l'API ArgoCD (RBAC `logs`).

Révéler la réponse

QCM d'ancrage (3/4)

Questions 5 à 6

5. Quels sont les quatre éléments d'une notification ?

A. Projet, dépôt, cluster, rôle

B. Trigger, template, service, abonnement

C. Hook, vague, phase, option

D. Source, destination, projet, politique

Réponse B — Trigger, template, service et abonnement.

Révéler la réponse

6. Quelle métrique remplace `argocd_app_health_status` depuis ArgoCD 3.0 ?

A. `argocd_app_reconcile`

B. `argocd_app_info` (label `health_status`)

C. `argocd_cluster_info`

D. `argocd_app_sync_total`

Réponse B — `argocd_app_info` et son label `health_status`.

Révéler la réponse

QCM d'ancrage (4/4)

Questions 7 à 7

7. Comment une application s'abonne-t-elle à une notification ?

A. Par une règle RBAC

B. Par une annotation `notifications.argoproj.io/subscribe.<trigger>.<service>`

C. Par un hook PostSync

D. Par une option de synchronisation

Réponse B — Annotation `notifications.argoproj.io/subscribe.<trigger>.<service>` .

Révéler la réponse

🎓 Module 8 : Déploiements bleu-vert et canari

Durée

1 h (S8.1 : 1 h)

Prérequis

M07 terminé (`flotte-web-dev` supervisée)

Fil rouge

Hexa Mobilité veut exposer une nouvelle version à une partie des utilisateurs avant de la généraliser

Objectifs

- Expliquer les stratégies bleu-vert et canari et leurs différences
- Remplacer un Deployment par un Rollout piloté par ArgoCD
- Conduire un canari : pauses, promotion, interruption depuis ArgoCD
- Conduire un bleu-vert avec service de prévisualisation et promotion manuelle

Déroulé

- Séquence S8.1 — Argo Rollouts avec ArgoCD (1 h)

Séquence S8.1

Argo Rollouts avec ArgoCD

1 h

1. Pourquoi changer de stratégie ?
2. Argo Rollouts en un schéma
3. Installer Argo Rollouts
4. Du Deployment au Rollout
5. Le canari étape par étape
6. Piloter le canari depuis ArgoCD
7. Bleu-vert : prévisualiser puis basculer
8. Analyse automatique du canari

❓ Pourquoi changer de stratégie ?

Le `RollingUpdate` remplace tout, sans point d'arrêt

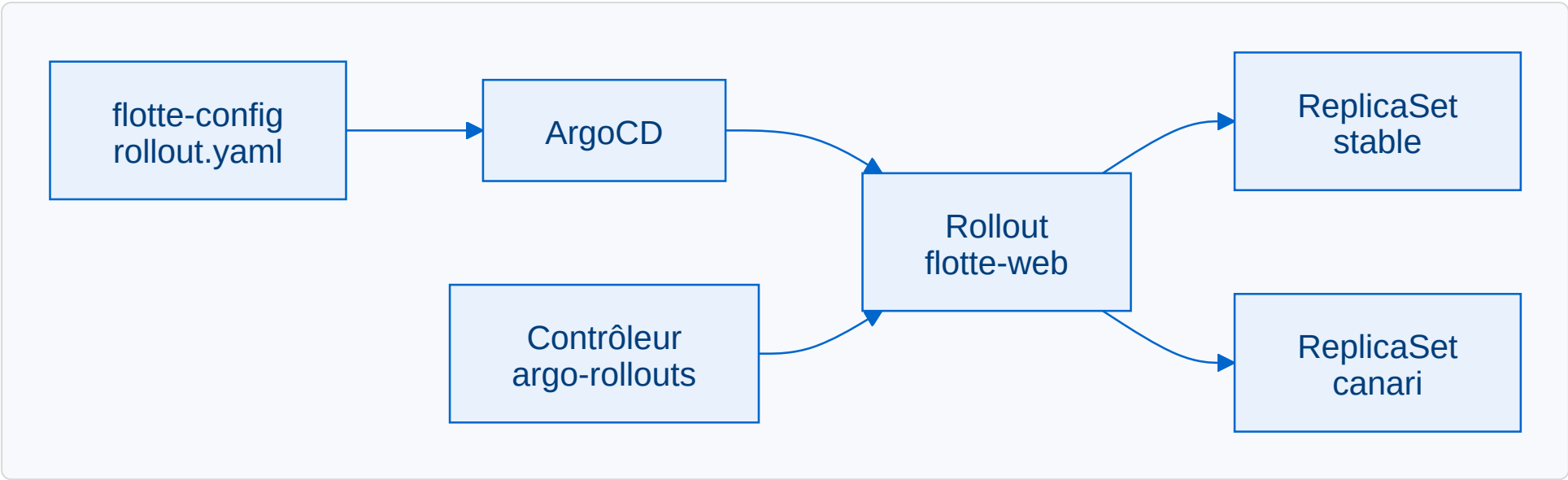
Livraison progressive = Exposition partielle + Vérification + Décision (promouvoir / interrompre)

STRATÉGIE	PRINCIPE	RETOUR ARRIÈRE
`RollingUpdate` (Deployment)	Remplacement progressif, sans pause	Nouveau déploiement
Canari	25 % → 50 % → 100 %, pauses et analyses	Immédiat (`abort`)
Bleu-vert	Nouvelle version complète à côté, bascule d'un coup	Immédiat (service actif inchangé)

- Outil : Argo Rollouts (projet Argo), contrôleur + CRD `Rollout`, `AnalysisTemplate`

🕮 Argo Rollouts en un schéma

ArgoCD applique le Rollout, Argo Rollouts conduit la livraison



- ArgoCD connaît la santé d'un Rollout : `Healthy`, `Progressing`, `Suspended` (pause), `Degraded`
- Actions ArgoCD sur un Rollout : `resume`, `promote-full`, `abort`, `retry`, `restart`

📖 Installer Argo Rollouts

Contrôleur dans le cluster, plugin `kubectl` sur le poste

```bash
kubectl create namespace argo-rollouts
kubectl apply -n argo-rollouts --server-side --force-conflicts \
  -f https://github.com/argoproj/argo-rollouts/releases/download/v1.10.2/install.yaml
kubectl argo rollouts version
```

Sortie attendue (dernière commande) :

```
kubectl-argo-rollouts: v1.10.2+be8a622
```

⚠ Piège courant

sans `--server-side`, CRD `rollouts` et `analysisruns` refusées (annotation > 262 Ko)

🕮 Du Deployment au Rollout

Même modèle de pod, `kind` et `strategy` changent

```yaml
apiVersion: argoproj.io/v1alpha1
kind: Rollout
metadata: {name: flotte-web}
spec:
  replicas: 4
  selector: {matchLabels: {app.kubernetes.io/name: flotte-web}}
  strategy:
    canary:
      steps: [{setWeight: 25}, {pause: {duration: 30s}},
              {setWeight: 50}, {pause: {}}, {setWeight: 100}]
  template: {}   # modèle de pod identique à celui du Deployment
```

- `deployment.yaml` supprimé de Git → Deployment `Pruned` par ArgoCD, Rollout créé
- Premier déploiement : 100 % directement (aucune version stable à comparer)

🕮 Le canari étape par étape

Sans routeur de trafic : poids = proportion de pods




```
Status:          ▫ Paused
Message:         CanaryPauseStep
  Step:          3/5
  SetWeight:     50
Images:          ghcr.io/stefanprodan/podinfo:6.16.0 (stable)
                 ghcr.io/stefanprodan/podinfo:6.17.0 (canary)
```

🕮 Piloter le canari depuis ArgoCD

Pause = application `Suspended` ; la synchronisation attend la fin du Rollout

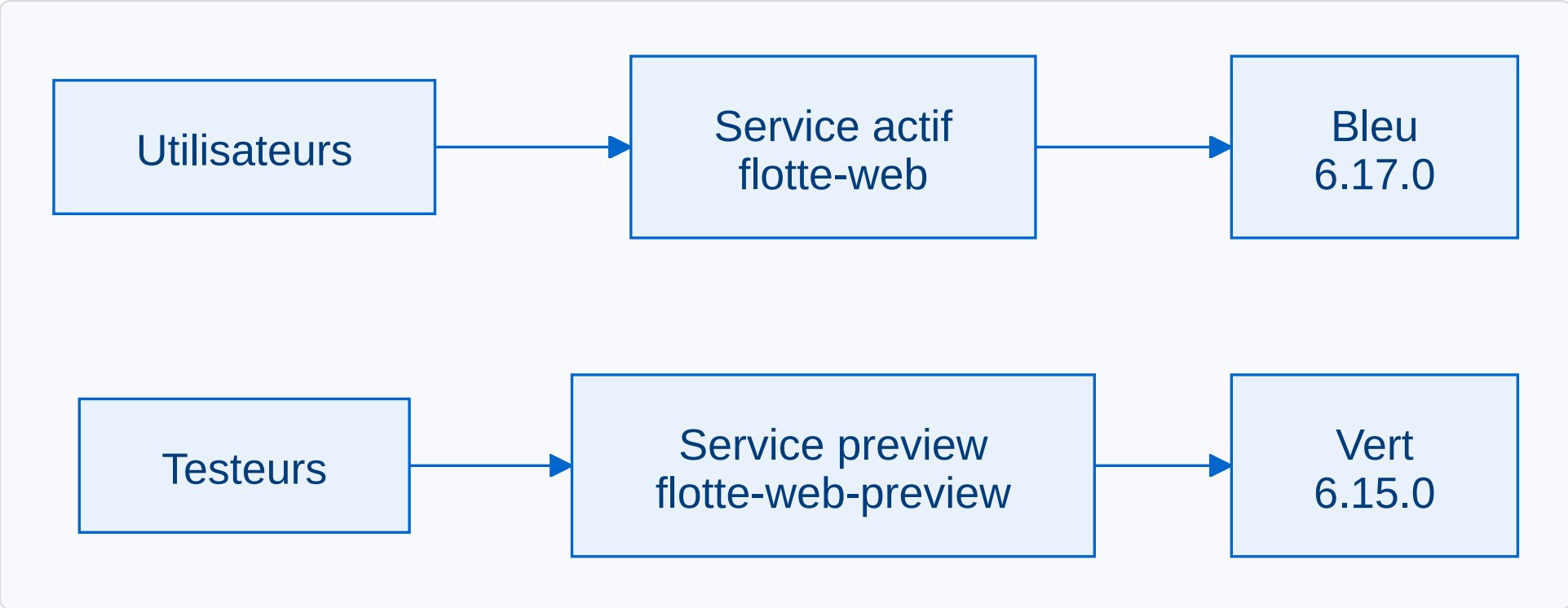
```bash
argocd app actions list flotte-web-dev --kind Rollout
argocd app actions run flotte-web-dev resume --kind Rollout --resource-name flotte-web
```

ACTION	EFFET
`resume`	Étape suivante
`promote-full`	100 % immédiatement, étapes sautées
`abort`	Retour à 0 % de canari, version stable conservée (`Degraded`)

- RBAC : `p, role:dev, applications, action/argoproj.io/Rollout/resume, flotte/*, allow`
- Hook PostSync (smoke test) exécuté APRÈS la promotion complète

🕮 Bleu-vert : prévisualiser puis basculer

Deux services, un seul reçoit le trafic réel



- `activeService`, `previewService`, `autoPromotionEnabled: false` : pause avant bascule
- Argo Rollouts réécrit le sélecteur des services (`rollouts-pod-template-hash`)
- Après promotion : ancienne version conservée `scaleDownDelaySeconds` (retour rapide)

🕮 Analyse automatique du canari

La décision promouvoir / interrompre confiée à une mesure

```yaml
strategy:
  canary:
    canaryService: flotte-web-canary
    steps: [{setWeight: 25}, {analysis: {templates: [{templateName: flotte-smoke}]}}]
```

Sortie observée (analyse en échec) :

```
RolloutAborted: Rollout aborted update to revision 6: Step-based analysis
phase error/failed: Metric "readyz" assessed Failed due to failed (2) > failureLimit (1)
```

- Fournisseurs : Job, Web, Prometheus, Datadog, New Relic (requête sur le canari)

À retenir

Points clés

- Rollout = Deployment + stratégie (canari ou bleu-vert) + pauses et analyses
- Installation des CRD Argo Rollouts : `--server-side` obligatoire (v1.10.2)
- Pause = `Suspended` dans ArgoCD ; actions `resume`, `promote-full`, `abort`
- Bleu-vert : service actif + service de prévisualisation, bascule manuelle
- Analyse automatique : interruption sans intervention humaine

QCM d'ancrage (1/4)

Questions 1 à 2

1. Quelle stratégie expose la nouvelle version à une fraction croissante des pods ?

- **A.** Bleu-vert
- **B.** Canari
- **C.** Recreate
- **D.** RollingUpdate sans pause

Réponse B — Le canari augmente progressivement la part de la nouvelle version.

Révéler la réponse

2. Quel statut de santé affiche ArgoCD pendant une pause du Rollout ?

- **A.** `Progressing`
- **B.** `Degraded`
- **C.** `Suspended`
- **D.** `Missing`

Réponse C — Un Rollout en pause est évalué `Suspended` par ArgoCD.

Révéler la réponse

QCM d'ancrage (2/4)

Questions 3 à 4

3. Que fait l'action `abort` sur un canari en cours ?

- **A.** Supprime le Rollout
- **B.** Ramène le trafic sur la version stable, canari à 0 %
- **C.** Promeut la nouvelle version
- **D.** Redémarre tous les pods

Réponse B — `abort` ramène le canari à 0 % ; la version stable continue de servir.

Révéler la réponse

4. Sans routeur de trafic, comment est réalisé un poids de 25 % sur 4 réplicas ?

- **A.** Par un Ingress pondéré
- **B.** Par 1 pod de la nouvelle version sur 4
- **C.** Par une règle de pare-feu
- **D.** C'est impossible

Réponse B — Sans routeur de trafic, le poids est la proportion de pods : 1 sur 4.

Révéler la réponse

QCM d'ancrage (3/4)

Questions 5 à 6

5. En bleu-vert, quel service reçoit le trafic des utilisateurs avant la promotion ?

- **A.** Le service de prévisualisation
- **B.** Le service actif, qui pointe sur l'ancienne version
- **C.** Les deux à parts égales
- **D.** Aucun

Réponse B — Le service actif reste sur l'ancienne version jusqu'à la promotion.

Révéler la réponse

6. Pourquoi l'installation d'Argo Rollouts v1.10.2 exige-t-elle `--server-side` ?

- **A.** Pour installer dans un namespace dédié
- **B.** Parce que certaines CRD dépassent la taille d'annotation de l'apply client
- **C.** Pour activer TLS
- **D.** Pour installer le plugin kubectl

Réponse B — Certaines CRD dépassent 262 144 octets d'annotation `last-applied`.

Révéler la réponse

QCM d'ancrage (4/4)

Questions 7 à 7

7. Quelle permission RBAC donne à l'équipe dev le droit de reprendre un canari en pause ?

A. `applications, sync`

B. `applications, action/argoproj.io/Rollout/resume`

C. `applications, update`

D. `projects, update`

Réponse B — Les actions de ressource sont autorisées par `action/<groupe>/<type>/<action>`.

Révéler la réponse

🎓 Module 9 : Helm et Kustomize

Durée

1 h 30 (S9.1 : 45 min · S9.2 : 45 min)

Prérequis

M08 terminé (`flotte-web` en Rollout)

Fil rouge

un composant `flotte-api` packagé par Helm, `flotte-web` décrit une fois et décliné en `dev`, `recette`, `prod`

Objectifs

- Déployer un chart Helm avec des valeurs stockées dans Git (multi-sources)
- Appliquer l'ordre de priorité des valeurs Helm dans ArgoCD
- Structurer une application en base Kustomize + overlays par environnement
- Générer une ConfigMap qui redémarre les pods à chaque changement

Déroulé

- Séquence S9.1 — Charts Helm avec ArgoCD (45 min)
- Séquence S9.2 — Kustomize : base et overlays (45 min)

Séquence S9.1

Charts Helm avec ArgoCD

45 min

1. Helm dans ArgoCD : rendu, pas installation
2. D'où vient le chart ?
3. Fournir les valeurs
4. Valeurs dans Git, chart à l'extérieur (1/2)
5. Valeurs dans Git, chart à l'extérieur (2/2)
6. Résultat : `flotte-api-dev`
7. Bonnes pratiques Helm avec ArgoCD

🕮 Helm dans ArgoCD : rendu, pas installation

ArgoCD utilise Helm comme moteur de gabarits

ArgoCD + Helm = helm template (repo-server) + apply (contrôleur)

ASPECT	`HELM INSTALL`	CHART VIA ARGOCD
Historique	Secrets de release Helm	Historique ArgoCD (M06)
`helm list`	Release visible	Vide
Retour arrière	`helm rollback`	`git revert` ou `app rollback`
Hooks Helm	Exécutés par Helm	Traduits en hooks ArgoCD
Dérive	Non détectée	Détectée, corrigée (self-heal)

❓ D'où vient le chart ?

Dépôt de charts, registre OCI ou dossier Git

```yaml
source:
  repoURL: https://stefanprodan.github.io/podinfo   # dépôt de charts HTTP
  chart: podinfo
  targetRevision: 6.17.0                             # version du chart
  helm:
    releaseName: flotte-api
```

- Registre OCI : `repoURL: <REGISTRE>/<CHEMIN>` sans `oci://` ; privé : Secret avec `enableOCI`
- Dossier Git contenant un `Chart.yaml` : `path` au lieu de `chart`
- Dépôt de charts autorisé dans `sourceRepos` du projet

🕮 Fournir les valeurs

Quatre mécanismes, un ordre de priorité

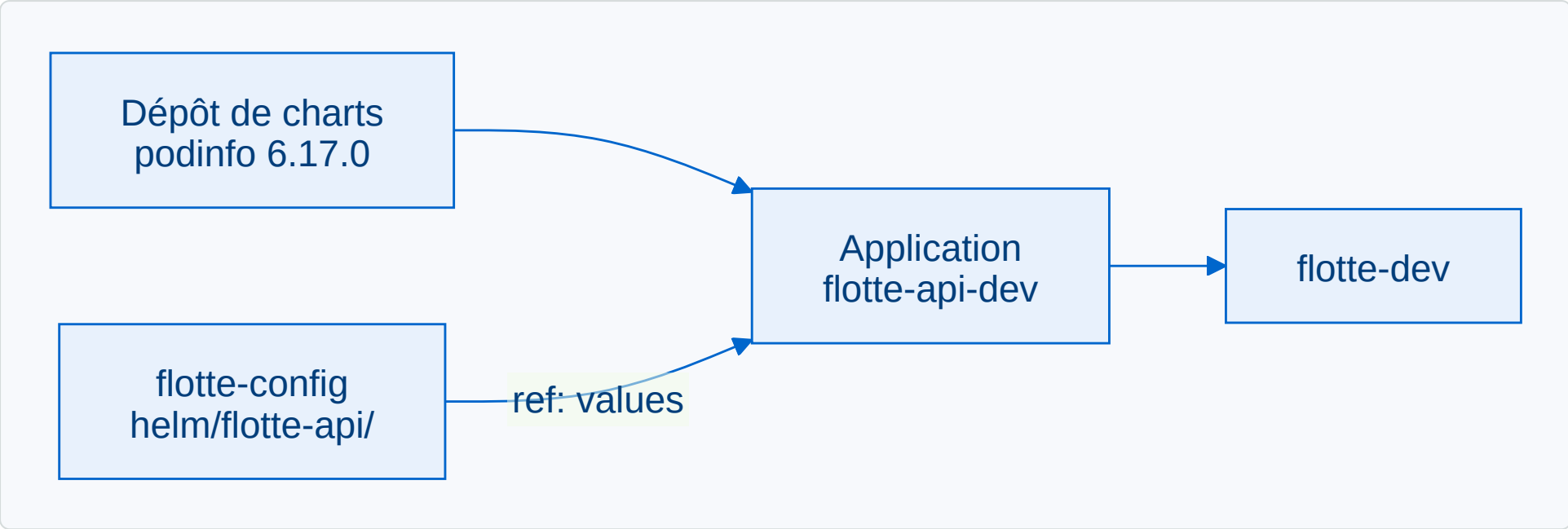
`parameters > valuesObject > values > valueFiles > values.yaml du chart`

CHAMP	FORME	USAGE
`valueFiles`	Fichiers (chart ou `$values`)	Valeurs par environnement, versionnées
`values`	Bloc de texte YAML	Historique, à éviter
`valuesObject`	Objet YAML validé	Quelques valeurs dans l'Application
`parameters`	Liste nom / valeur	Surcharge ponctuelle (`--helm-set`)

- Testé : `valueFiles` (1) < `valuesObject` (3) < `parameters` (2) → 2 réplicas

Valeurs dans Git, chart à l'extérieur (1/2)

Application multi-sources : une source de référence `$values`



- Chart : version figée chez l'éditeur, jamais copié dans le dépôt
- Valeurs : `flotte-config/helm/flotte-api/`, revues comme le reste de la configuration
- Un commit sur les valeurs OU une nouvelle version de chart déclenche la synchronisation

Valeurs dans Git, chart à l'extérieur (2/2)

`argocd/flotte-api-dev.yaml` (extrait)

```yaml
sources:
  - repoURL: https://stefanprodan.github.io/podinfo
    chart: podinfo
    targetRevision: 6.17.0
    helm: {valueFiles: [$values/helm/flotte-api/values-common.yaml,
                        $values/helm/flotte-api/values-dev.yaml]}
  - repoURL: https://gitlab.com/<GITLAB_USER>/flotte-config.git
    targetRevision: main
    ref: values
```

Résultat : `flotte-api-dev`

Rendu vérifié avant synchronisation

```bash
argocd app manifests flotte-api-dev | grep -E 'replicas:|UI_MESSAGE|UI_COLOR|image:' -A1
```

Sortie attendue (extrait) :

```
  replicas: 1
        - name: PODINFO_UI_MESSAGE
          value: Hexa Mobilite - API
        - name: PODINFO_UI_COLOR
          value: '#2e7d32'
        image: ghcr.io/stefanprodan/podinfo:6.17.0
```

- `Sync Status: Synced to 6.17.0` : révision = version du chart

Bonnes pratiques Helm avec ArgoCD

Reproductible, lisible, testable

- Version de chart figée (`6.17.0`), jamais `*` ni une plage
- Un fichier de valeurs commun + un par environnement, dans Git
- Tester le rendu en local : `helm template flotte-api podinfo --repo <URL> -f values-dev.yaml`
- Charts internes publiés dans un registre OCI (versionnés, signés)
- `parameters` réservés aux surcharges temporaires (non tracées dans Git)

⚠ Piège courant

`valueFiles` dans l'ordre alphabétique avec un joker (`*.yaml`) : le dernier fichier gagne

Séquence S9.2

Kustomize : base et overlays

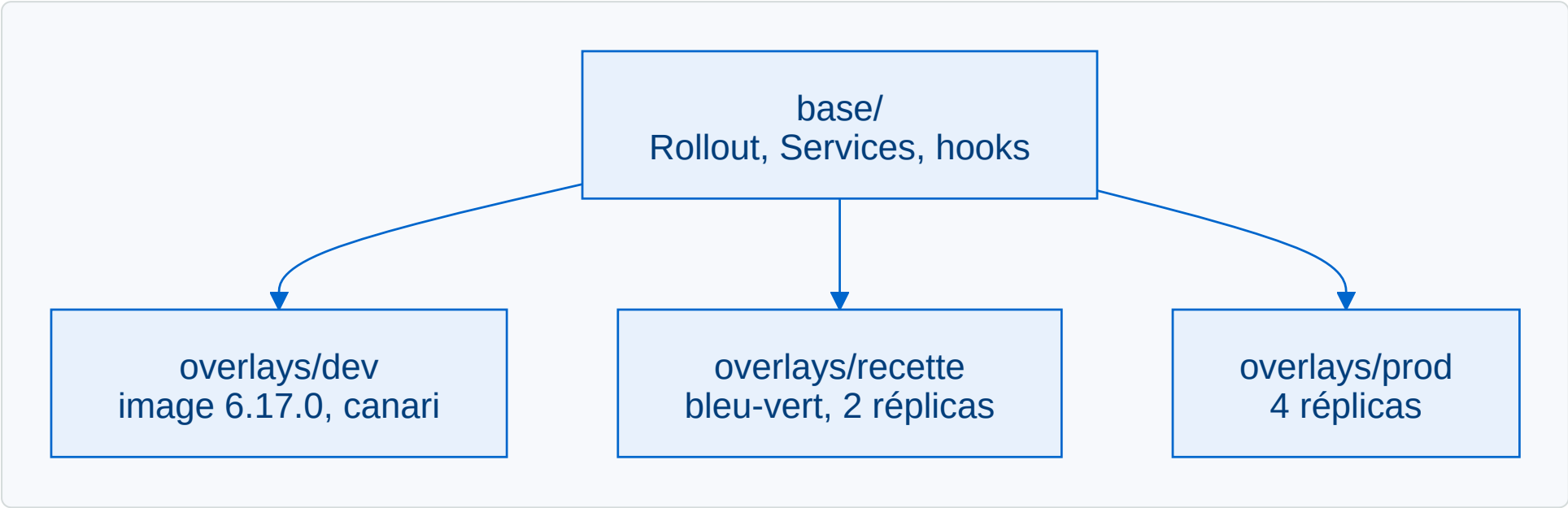
45 min

1. Une base, des variantes
2. La base
3. Un overlay : `dev`
4. Un patch : la recette en bleu-vert
5. La ConfigMap générée
6. Anciennes ConfigMaps : conservées, ignorées
7. Helm ou Kustomize ?

🕮 Une base, des variantes

Décrire une fois, varier par environnement

`Overlay = Base + écarts explicites de l'environnement`



- Une Application par overlay : `path: apps/flotte-web/overlays/dev`
- Avant : `apps/flotte-web/` et `apps/flotte-web-recette/` dupliqués (dérive entre copies)

La base

`apps/flotte-web/base/kustomization.yaml` (extrait)

```yaml
configurations: [rollout-transform.yaml]   # Kustomize comprend le Rollout
resources: [rollout.yaml, service.yaml, service-canary.yaml, analysis-template.yaml,
            hook-presync-migration.yaml, hook-postsync-smoke-test.yaml]
configMapGenerator:
  - name: flotte-web-config
    literals: ["PODINFO_UI_MESSAGE=Hexa Mobilite - suivi de flotte"]
images:
  - {name: ghcr.io/stefanprodan/podinfo, newTag: 6.16.0}
```

- `rollout-transform.yaml` : fourni par Argo Rollouts (version Kustomize 5)
- Sans lui : références de ConfigMap non réécrites dans le Rollout

📖 Un overlay : `dev`

Seuls les écarts sont écrits

YAML

```yaml
resources: [../../base]
configMapGenerator:
  - name: flotte-web-config
    behavior: merge
    literals: ["PODINFO_UI_MESSAGE=Hexa Mobilite - DEV"]
images:
  - {name: ghcr.io/stefanprodan/podinfo, newTag: 6.17.0}
```

BASH

```bash
kubectl kustomize apps/flotte-web/overlays/dev | grep -E 'image: ghcr|config-'
```

Sortie attendue :

```
    name: flotte-web-config-4c48ggb2h2
  image: ghcr.io/stefanprodan/podinfo:6.17.0
```

🕮 Un patch : la recette en bleu-vert

Remplacer une partie de la base, sans la copier

```yaml
resources: [../../base, service-preview.yaml]
replicas: [{name: flotte-web, count: 2}]
patches:
  - target: {group: argoproj.io, kind: Rollout, name: flotte-web}
    patch: |-
      - op: replace
        path: /spec/strategy
        value: {blueGreen: {activeService: flotte-web,
                previewService: flotte-web-preview, autoPromotionEnabled: false}}
```

- Patch JSON (`op` , `path`) : précis sur une ressource personnalisée
- `replicas` : transformateur Kustomize (Rollout reconnu grâce à `rollout-transform.yaml`)

📖 La ConfigMap générée

Nom haché = nouveau nom à chaque changement = pods redémarrés

🔁 Avant / Après

Avant (M06): message modifié dans la ConfigMap → pods inchangés, ancien message servi
Après (générateur): `flotte-web-config-4c48ggb2h2` → `flotte-web-config-fckt6km9b8` → nouveaux pods

```yaml
options:
  annotations:
    argocd.argoproj.io/sync-wave: "-1"
    argocd.argoproj.io/sync-options: Prune=false
    argocd.argoproj.io/compare-options: IgnoreExtraneous
```

- Sans `Prune=false` : ancienne ConfigMap supprimée alors que la version stable l'utilise

📖 Anciennes ConfigMaps : conservées, ignorées

Comportement testé sur `flotte-web-dev`

```
ConfigMap  flotte-dev  flotte-web-config-4c48ggb2h2  OutOfSync  ignored (no prune)
ConfigMap  flotte-dev  flotte-web-config-fckt6km9b8  Synced     created
```

- Application `Synced` malgré l'ancienne ConfigMap (`IgnoreExtraneous`)
- Retour arrière ou `abort` d'un canari : la version stable retrouve sa configuration
- Nettoyage périodique des anciennes ConfigMaps (script ou politique d'équipe)

? Helm ou Kustomize ?

Complémentaires plus que concurrents

CRITÈRE	HELM	KUSTOMIZE
Point de départ	Chart paramétré (gabarits)	YAML brut + patchs
Logiciel tiers	Format standard des éditeurs	Rare
Variantes par environnement	Fichiers de valeurs	Overlays
Lisibilité du rendu	Gabarits Go	Proche du YAML final

- Combinaison : chart tiers en Helm, applications internes en Kustomize
- Chart inclus dans Kustomize (`helmCharts`) : `kustomize.buildOptions: --enable-helm` dans `argocd-cm`

À retenir

Points clés

- Helm dans ArgoCD = rendu (`helm template`), cycle de vie géré par ArgoCD
- Valeurs : `parameters > valuesObject > values > valueFiles` ; `$values` pour des valeurs dans Git
- Kustomize : base + overlays, une Application par environnement
- `configMapGenerator` : changement de configuration = pods redémarrés
- ConfigMaps générées : `Prune=false` + `IgnoreExtraneous` (documentation ArgoCD)

QCM d'ancrage (1/4)

Questions 1 à 2

1. Pourquoi `helm list` est-il vide pour une application Helm déployée par ArgoCD ?

- **A.** Le chart est mal installé
- **B.** ArgoCD rend le chart (`helm template`) sans créer de release Helm
- **C.** Helm n'est pas installé dans le cluster
- **D.** Le namespace est différent

Réponse B — ArgoCD exécute l'équivalent de `helm template` , aucune release Helm n'est créée.

Révéler la réponse

2. Une valeur est définie dans `valueFiles` , `valuesObject` et `parameters` . Laquelle s'applique ?

- **A.** `valueFiles`
- **B.** `valuesObject`
- **C.** `parameters`
- **D.** Celle du `values.yaml` du chart

Réponse C — `parameters` a la priorité la plus haute.

Révéler la réponse

QCM d'ancrage (2/4)

Questions 3 à 4

3. À quoi sert une source avec `ref: values` ?

- **A.** À désigner un dépôt dont les fichiers sont référencés par `$values/...`
- **B.** À stocker les valeurs dans un Secret
- **C.** À fusionner deux charts
- **D.** À choisir la branche du chart

Réponse A — `ref: values` expose le dépôt aux autres sources sous `$values/`.

Révéler la réponse

4. Dans un overlay Kustomize, que contient le fichier `kustomization.yaml` ?

- **A.** Une copie complète des manifestes
- **B.** Une référence à la base et les seuls écarts de l'environnement
- **C.** Les valeurs Helm
- **D.** La définition de l'Application ArgoCD

Réponse B — Un overlay référence la base et décrit uniquement ses écarts.

Révéler la réponse

QCM d'ancrage (3/4)

Questions 5 à 6

5. Pourquoi un `configMapGenerator` redémarre-t-il les pods à chaque changement ?

- **A.** Il envoie un signal aux pods
- **B.** Le nom de la ConfigMap change (empreinte), donc le modèle de pod change
- **C.** ArgoCD force un `rollout restart`
- **D.** Kubernetes surveille le contenu des ConfigMaps

Réponse B — L'empreinte du contenu change le nom, donc le modèle de pod, donc un nouveau déploiement.

Révéler la réponse

6. Pourquoi ajouter `rollout-transform.yaml` à la base ?

- **A.** Pour installer Argo Rollouts
- **B.** Pour que Kustomize réécrive références de ConfigMap, images et réplicas dans un Rollout
- **C.** Pour convertir le Rollout en Deployment
- **D.** Pour activer le canari

Réponse B — `rollout-transform.yaml` décrit à Kustomize les chemins d'un Rollout (références, images, réplicas).

Révéler la réponse

QCM d'ancrage (4/4)

Questions 7 à 7

7. Pourquoi marquer les ConfigMaps générées `Prune=false` ?

A. Pour accélérer la synchronisation

B. Pour conserver la configuration utilisée par la version stable (retour arrière)

C. Pour masquer les ConfigMaps dans l'UI

D. Parce que les ConfigMaps ne peuvent pas être supprimées

Réponse B — La version stable utilise encore l'ancienne ConfigMap pendant un canari ou après un retour arrière.

Révéler la réponse

Module 10 : Intégration continue avec ArgoCD

Durée

1 h 15 (S10.1 : 1 h 15)

Prérequis

M09 terminé (overlay `dev` en Kustomize)

Fil rouge

chaque commit de `flotte-app` doit arriver en `dev` sans intervention

Objectifs

- Répartir les rôles entre CI (construire) et ArgoCD (déployer)
- Écrire un pipeline GitLab CI qui publie une image et met à jour `flotte-config`
- Configurer un webhook pour une détection immédiate des commits
- Transposer le pipeline à Jenkins et comparer les variantes d'intégration

Déroulé

- Séquence S10.1 — La CI écrit dans Git, ArgoCD déploie (1 h 15)

Séquence S10.1

La CI écrit dans Git, ArgoCD déploie

1 h 15

1. Qui fait quoi ?
2. Le flux complet
3. Deux dépôts, deux rôles
4. Pipeline GitLab CI : construire
5. Pipeline GitLab CI : mettre à jour la configuration
6. Webhook : déployer sans attendre
7. Le même pipeline avec Jenkins
8. Autres modèles d'intégration

❓ Qui fait quoi ?

La CI produit l'artefact, ArgoCD le déploie

`CI = Construire + Tester + Publier l'image + Mettre à jour la configuration`

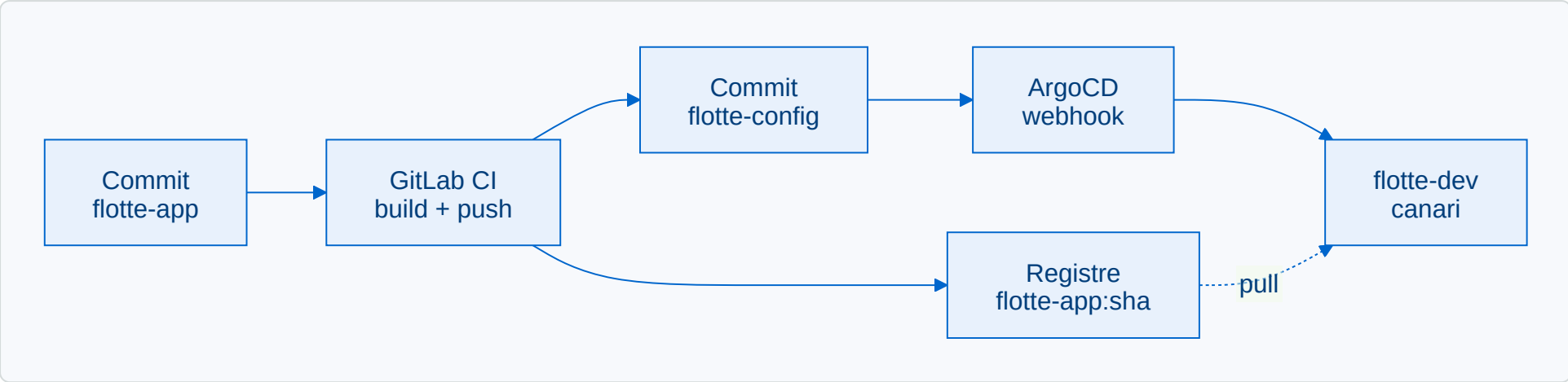
ÉTAPE	OUTIL	ACCÈS NÉCESSAIRE
Construire, tester	GitLab CI, Jenkins	Code de `flotte-app`
Publier l'image	GitLab CI, Jenkins	Registre d'images
Modifier le tag dans `flotte-config`	GitLab CI, Jenkins	Écriture sur `flotte-config`
Déployer	ArgoCD	Cluster (depuis l'intérieur)

🏢 En entreprise

la CI ne détient AUCUN accès au cluster (identifiants hors cluster supprimés)

🕮 Le flux complet

Du commit applicatif au pod



- Tag d'image = `$CI_COMMIT_SHORT_SHA` : chaque déploiement remonte à un commit de code
- Historique de `flotte-config` = journal des déploiements

📖 Deux dépôts, deux rôles

Code et configuration séparés

	`FLOTTE-APP`	`FLOTTE-CONFIG`
Contenu	Code, `Dockerfile`, `.gitlab-ci.yml`	Manifestes, overlays, Applications
Déclencheur	Commit des développeurs	Commit de la CI ou d'un humain
Lu par	Runners CI	ArgoCD (jeton de déploiement, lecture)
Écrit par	Développeurs	CI (jeton dédié), équipe ops (merge request)

- Pas de boucle : un commit de configuration ne relance pas la construction de l'image
- Promotion `recette` / `prod` : merge request sur `flotte-config` (revue humaine)

∞ Pipeline GitLab CI : construire

Image publiée dans le registre du projet

```yaml
stages: [build, update-config]
build-image:
  stage: build
  image: docker:28
  services: [docker:28-dind]
  variables: {DOCKER_TLS_CERTDIR: "/certs"}
  script:
    - echo "$CI_REGISTRY_PASSWORD" |
        docker login -u "$CI_REGISTRY_USER" --password-stdin "$CI_REGISTRY"
    - docker build --build-arg GIT_SHA="$CI_COMMIT_SHA" -t "$IMAGE" .
    - docker push "$IMAGE"
```

- `IMAGE: $CI_REGISTRY_IMAGE:$CI_COMMIT_SHORT_SHA` (variable globale du pipeline)
- Variables `CI_REGISTRY_*`, `CI_COMMIT_*` : prédéfinies par GitLab

∞ Pipeline GitLab CI : mettre à jour la configuration

Un commit dans `flotte-config`, rien d'autre

```yaml
update-config:
  stage: update-config
  image: alpine:3.22
  rules: [{if: $CI_COMMIT_BRANCH == $CI_DEFAULT_BRANCH}]
  variables:
    CONFIG_REPO_URL: $FLOTTE_CONFIG_URL
    CONFIG_TOKEN: $FLOTTE_CONFIG_TOKEN
    NEW_IMAGE: $IMAGE
  before_script: [apk add --no-cache git kustomize]
  script: [sh ci/update-config.sh]
```

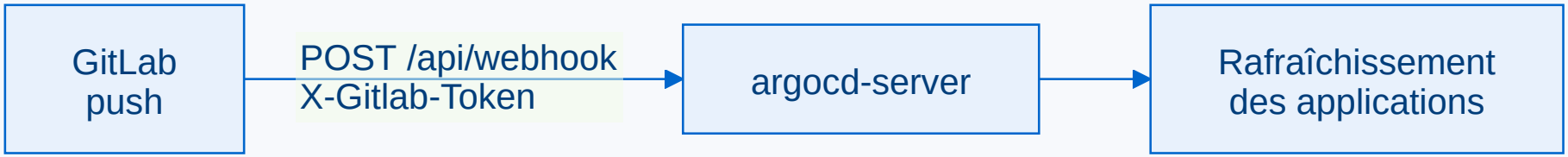
Sortie du script (testée) :

```
flotte-config updated: ghcr.io/stefanprodan/podinfo:6.15.0
```

- Cœur du script : `kustomize edit set image ghcr.io/stefanprodan/podinfo=$NEW_IMAGE`

🕮 Webhook : déployer sans attendre

Détection immédiate au lieu de 120 s + jitter



- GitLab : `flotte-config` → *Settings* → *Webhooks* → URL `https://<ARGOCD>/api/webhook`, événement *Push*
- ArgoCD : clé `webhook.gitlab.secret` dans `argocd-secret` = jeton secret du webhook
- Cluster `kind` local : injoignable depuis GitLab.com → réconciliation ou *Refresh* en formation

⚠ Piège courant

webhook sans secret = n'importe qui peut déclencher des rafraîchissements

∞ Le même pipeline avec Jenkins

`Jenkinsfile` déclaratif (démonstration)

```groovy
stage('Update flotte-config') {
  when { branch 'main' }
  steps {
    withCredentials([string(credentialsId: 'flotte-config-token', variable: 'TOKEN')]) {
      sh '''CONFIG_REPO_URL="$FLOTTE_CONFIG_URL" CONFIG_TOKEN="$TOKEN" \
            NEW_IMAGE="$IMAGE" sh ci/update-config.sh'''
    }
  }
}
```

- Même script `ci/update-config.sh` : la logique GitOps ne dépend pas de l'outil de CI
- Identifiants : magasin de credentials Jenkins (`withCredentials`), jamais dans le `Jenkinsfile`
- `when { branch 'main' }` : job *Multibranch Pipeline* uniquement (étape sautée dans un job *Pipeline* simple)

Autres modèles d'intégration

Choisir selon le réseau et le niveau de contrôle

MODÈLE	PRINCIPE	LIMITE
CI écrit dans Git (ce module)	Commit du nouveau tag	Une écriture par livraison
CI appelle ArgoCD	`argocd app sync`, `app wait` avec jeton `ci-gitlab` (M03)	API ArgoCD joignable depuis la CI
Argo CD Image Updater	Surveille le registre, met à jour le tag	Composant supplémentaire
Promotion par merge request	Recette, prod : revue humaine	Délai de revue

- `argocd app wait --health` en fin de pipeline : la CI rapporte le résultat du déploiement

À retenir

Points clés

- CI = construire, publier, écrire le tag dans `flotte-config` ; ArgoCD = déployer
- Aucun identifiant du cluster dans la CI ; jeton d'écriture limité au dépôt de configuration
- Tag d'image = SHA du commit : traçabilité code → image → déploiement
- Webhook + secret : détection immédiate ; sinon réconciliation périodique
- Même script pour GitLab CI et Jenkins

QCM d'ancrage (1/4)

Questions 1 à 2

1. Dans le modèle pull, que modifie le pipeline CI pour déployer une nouvelle version ?

- **A.** Le Deployment dans le cluster
- **B.** Le tag d'image dans le dépôt de configuration
- **C.** L'Application ArgoCD par l'API
- **D.** Le Secret de dépôt

Réponse B — La CI modifie le tag d'image dans `flotte-config` ; ArgoCD applique.

Révéler la réponse

2. Pourquoi utiliser le SHA du commit comme tag d'image ?

- **A.** Pour réduire la taille de l'image
- **B.** Pour relier chaque déploiement à un commit de code (traçabilité)
- **C.** Parce que `latest` est interdit par Kubernetes
- **D.** Pour accélérer le téléchargement

Réponse B — Le SHA relie l'image déployée au commit de code.

Révéler la réponse

QCM d'ancrage (2/4)

Questions 3 à 4

3. Quel accès la CI doit-elle détenir dans ce modèle ?

- **A.** Un kubeconfig administrateur
- **B.** Le mot de passe `admin` d'ArgoCD
- **C.** L'écriture sur `flotte-config` et sur le registre d'images
- **D.** Aucun accès

Réponse C — Écriture sur le dépôt de configuration et sur le registre ; aucun accès au cluster.

Révéler la réponse

4. Quel est l'intérêt du webhook GitLab vers ArgoCD ?

- **A.** Chiffrer les manifestes
- **B.** Déclencher la détection immédiate d'un commit
- **C.** Remplacer la synchronisation automatique
- **D.** Construire l'image

Réponse B — Le webhook déclenche le rafraîchissement immédiat des applications concernées.

Révéler la réponse

QCM d'ancrage (3/4)

Questions 5 à 6

5. Où se configure le secret du webhook GitLab côté ArgoCD ?

- **A.** `argocd-cm`, clé `webhook.gitlab`
- **B.** `argocd-secret`, clé `webhook.gitlab.secret`
- **C.** Annotation de l'Application
- **D.** Variable de la CI

Réponse B — Clé `webhook.gitlab.secret` du Secret `argocd-secret`.

Révéler la réponse

6. Quelle commande Kustomize modifie le tag d'image d'un overlay ?

- **A.** `kustomize build --image`
- **B.** `kustomize edit set image <NOM>=<NOUVEAU_NOM>:<TAG>`
- **C.** `kubectl set image`
- **D.** `argocd app set --image`

Réponse B — `kustomize edit set image <NOM>=<NOUVEAU_NOM>:<TAG>`.

Révéler la réponse

QCM d'ancrage (4/4)

Questions 7 à 7

7. Pourquoi un commit sur `flotte-config` ne relance-t-il pas la construction de l'image ?

A. Parce que GitLab bloque les boucles

B. Parce que code et configuration sont dans deux dépôts distincts

C. Parce qu'ArgoCD annule le pipeline

D. Parce que le webhook est désactivé

Réponse B — Deux dépôts distincts : le pipeline de `flotte-app` n'est pas déclenché par `flotte-config`.

Révéler la réponse

🎓 Module 11 : Sécurité, identité et secrets

Durée
1 h 15 (S11.1 : 35 min · S11.2 : 40 min)

Prérequis
M10 terminé, comptes et RBAC du M03

Fil rouge
Hexa Mobilité prépare l'audit de sécurité de sa plateforme GitOps

Objectifs

- Relier identité, RBAC, AppProjects et droits Kubernetes en défense en profondeur
- Déléguer l'identité à un fournisseur OIDC et attribuer des droits par groupe ou par projet
- Durcir une installation d'ArgoCD et identifier ses points d'extension
- Stocker un secret chiffré dans Git avec Sealed Secrets et le consommer

Déroulé

- Séquence S11.1 — Identité, autorisations, durcissement (35 min)
- Séquence S11.2 — Secrets dans un dépôt GitOps (40 min)

Séquence S11.1

Identité, autorisations, durcissement

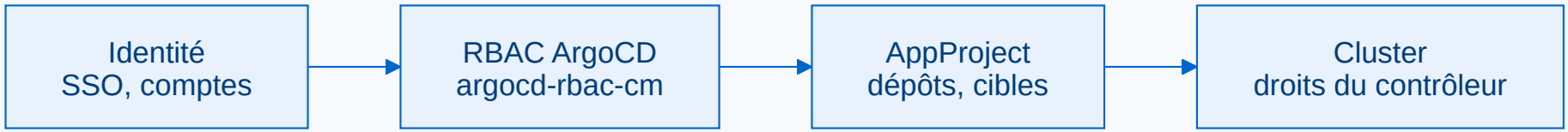
35 min

1. Les couches de sécurité
2. Identité : déléguer au fournisseur de l'entreprise
3. Autorisations par projet
4. Durcir l'installation
5. Points d'extension

Les couches de sécurité

Chaque couche limite la précédente

Accès effectif = Identité ∩ RBAC ArgoCD ∩ AppProject ∩ Droits Kubernetes du contrôleur



- M03 : comptes locaux, RBAC, projet `flotte` ; ce module : SSO, rôles de projet, secrets
- Une faille dans une couche ne donne pas tout : principe de défense en profondeur

🕮 Identité : déléguer au fournisseur de l'entreprise

OIDC direct ou Dex (connecteurs LDAP, SAML, GitLab, GitHub)

```yaml
data:
  url: https://argocd.example.com
  oidc.config: |
    name: Annuaire Hexa Mobilite
    issuer: https://<IDP>/realms/hexa
    clientID: argocd
    clientSecret: $oidc.flotte.clientSecret
    requestedScopes: ["openid", "profile", "email", "groups"]
```

```
g, hexa:equipe-dev, role:dev
g, hexa:equipe-ops, role:admin
```

- Groupes de l'annuaire → rôles ArgoCD : arrivée et départ gérés dans l'annuaire

🕮 Autorisations par projet

Rôles et jetons propres à un AppProject

```bash
argocd proj role create flotte ci-deployer
argocd proj role add-policy flotte ci-deployer \
  --action sync --permission allow --object 'flotte-*'
argocd proj role create-token flotte ci-deployer --expires-in 168h --token-only
```

Sortie attendue (`argocd proj role get flotte ci-deployer`, extrait) :

```
p, proj:flotte:ci-deployer, applications, sync, flotte/flotte-*, allow
```

- Délégation : l'équipe propriétaire du projet gère ses jetons, sans toucher au RBAC global
- Jetons suivis dans `status.jwtTokensByRole` ; révocation : `argocd proj role delete-token`

🕮 Durcir l'installation

Liste de contrôle avant la production

MESURE	RÉGLAGE	DÉFAUT
Compte `admin` désactivé	`admin.enabled: "false"`	Activé
Accès anonyme interdit	`users.anonymous.enabled: "false"`	Désactivé
Terminal web fermé	`exec.enabled: "false"`	Désactivé
Sessions courtes	`users.session.duration: "8h"`	24 h
TLS reconnu	Certificat dans `argocd-server-tls`	Auto-signé
Rollback séparé de sync	`server.rbac.rollback.enforce.enable` (M06)	Non

- Conservés : NetworkPolicies du manifeste, `policy.default` vide, dépôts *scopés* par projet

📖 Points d'extension

Adapter ArgoCD sans le modifier

BESOIN	EXTENSION	CONFIGURATION
Outil de rendu non natif	Plugin de configuration (CMP)	Sidecar du `repo-server`
Santé d'une ressource personnalisée	Script Lua	`resource.customizations.health.*`
Bouton d'action sur une ressource	Action Lua	`resource.customizations.actions.*`
Vue spécifique dans l'UI	Extension d'UI et proxy	`extension.config` (+ `server.enable.proxy.extension`)

- Rollout (M08), SealedSecret : santé et actions déjà fournies par ArgoCD
- Plugin CMP = commandes exécutées dans le `repo-server` : revue, version figée

Séquence S11.2

Secrets dans un dépôt GitOps

40 min

1. Le problème des secrets
2. Sealed Secrets en un schéma
3. Sceller un secret
4. Consommer le secret
5. Portée et exploitation
6. External Secrets Operator

⚠ Le problème des secrets

Tout est dans Git… sauf ce qui ne doit jamais y être

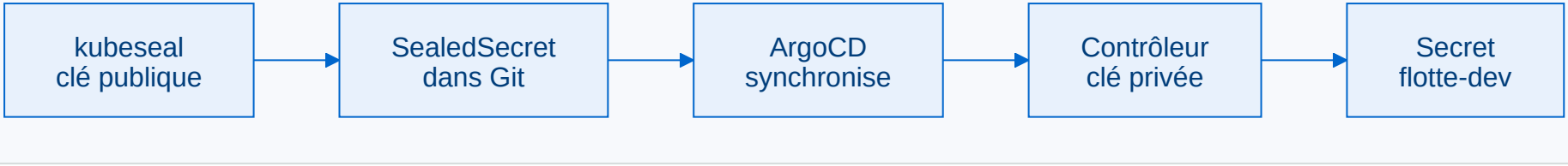
⇄ Avant / Après

Avant: `Secret` en base64 dans Git → lisible par quiconque clone le dépôt (base64 ≠ chiffrement)
Après: secret CHIFFRÉ dans Git, déchiffré uniquement dans le cluster

APPROCHE	PRINCIPE	OÙ VIT LA VALEUR
Sealed Secrets	Chiffrement asymétrique, contrôleur dans le cluster	Git (chiffrée)
External Secrets Operator	Lecture d'un coffre (Vault, cloud)	Coffre externe
SOPS (plugin)	Fichiers chiffrés par clé KMS ou age	Git (chiffrée)

🕮 Sealed Secrets en un schéma

Clé publique pour chiffrer, clé privée dans le cluster



- Contrôleur v0.41.2 dans `kube-system` (`controller.yaml` de la version)
- Portée par défaut *strict* : un secret scellé = un nom + un namespace

🕮 Sceller un secret

Le secret en clair ne quitte jamais le poste

```bash
kubectl create secret generic flotte-web-secrets -n flotte-dev \
  --from-literal=FLOTTE_API_KEY=<VALEUR> --dry-run=client -o yaml \
  | kubeseal --format yaml > apps/flotte-web/overlays/dev/sealed-secret.yaml
```

Sortie attendue (extrait du fichier) :

```
kind: SealedSecret
metadata:
  name: flotte-web-secrets
  namespace: flotte-dev
spec:
  encryptedData:
    FLOTTE_API_KEY: AgBZXuRzoWpbNHFbec5C6bpT/2oJN6yKqtlY5bUQhkkyF6baoXX
```

- `--dry-run=client` : aucun Secret créé dans le cluster, seul le fichier chiffré est produit

🕮 Consommer le secret

Le Rollout lit le Secret créé par le contrôleur

```yaml
envFrom:
  - configMapRef: {name: flotte-web-config}
  - secretRef: {name: flotte-web-secrets, optional: true}
```

Résultat testé (après synchronisation et promotion) :

```
bitnami.com  SealedSecret  flotte-dev  flotte-web-secrets  Synced  Healthy
flotte-web-85675c5597-r777v FLOTTE_API_KEY=hexa-dev-0001
```

- `optional: true` dans la base : `recette` et `prod` démarrent sans ce secret
- Secret créé avec un `ownerReference` vers le SealedSecret (supprimé avec lui)

🕮 Portée et exploitation

Ce qui protège… et ce qui peut tout casser

```
no key could decrypt secret (FLOTTE_API_KEY)
```

(même SealedSecret appliqué dans `flotte-recette` : refusé, portée *strict*)

- Clé privée du contrôleur : à sauvegarder HORS du cluster (perte = secrets indéchiffrables)
- Clé renouvelée tous les 30 jours, anciennes conservées ; rescellement périodique conseillé
- Un secret par environnement : sceller pour `flotte-recette`, `flotte-prod` séparément

⚠ Piège courant

cluster recréé sans restaurer la clé = tous les SealedSecrets inutilisables

📖 External Secrets Operator

La valeur reste dans le coffre, Git ne contient qu'une référence

```yaml
apiVersion: external-secrets.io/v1
kind: ExternalSecret
metadata: {name: flotte-web-secrets}
spec:
  refreshInterval: 1h
  secretStoreRef: {kind: ClusterSecretStore, name: coffre-hexa}
  target: {name: flotte-web-secrets}
  data:
    - secretKey: FLOTTE_API_KEY
      remoteRef: {key: flotte/dev, property: api-key}
```

- Rotation dans le coffre, propagée sans commit ; adapté quand un coffre existe déjà

À retenir

Points clés

- Accès effectif = identité ∩ RBAC ∩ AppProject ∩ droits du contrôleur
- SSO : groupes de l'annuaire → rôles ; rôles de projet pour déléguer (jetons CI)
- Durcissement : `admin` off, anonyme off, exec off, TLS, sessions courtes
- Jamais de `Secret` en clair dans Git : Sealed Secrets (Git chiffré) ou ESO (coffre)
- Clé privée de Sealed Secrets sauvegardée hors du cluster

QCM d'ancrage (1/4)

Questions 1 à 2

1. Pourquoi un `Secret` Kubernetes en base64 dans Git est-il une fuite ?

- **A.** Base64 est un chiffrement faible
- **B.** Base64 est un encodage, réversible sans clé
- **C.** Git refuse les fichiers base64
- **D.** Ce n'est pas une fuite

Réponse B — Base64 est un encodage réversible sans clé.

Révéler la réponse

2. Avec Sealed Secrets, où se trouve la clé capable de déchiffrer ?

- **A.** Dans Git
- **B.** Sur le poste du développeur
- **C.** Dans le cluster, gérée par le contrôleur
- **D.** Dans ArgoCD

Réponse C — La clé privée est détenue par le contrôleur, dans le cluster.

Révéler la réponse

QCM d'ancrage (2/4)

Questions 3 à 4

3. Un SealedSecret scellé pour `flotte-dev` est appliqué dans `flotte-recette`. Résultat ?

- **A.** Le Secret est créé dans `flotte-recette`
- **B.** Refus : `no key could decrypt secret` (portée strict)
- **C.** Le Secret est copié dans les deux namespaces
- **D.** ArgoCD le chiffre de nouveau

Réponse B — Portée strict : nom et namespace sont liés au chiffrement.

Révéler la réponse

4. Quel est l'intérêt d'un rôle d'AppProject avec jeton ?

- **A.** Remplacer le SSO
- **B.** Déléguer des droits limités à un projet (ex. pipeline CI)
- **C.** Chiffrer les secrets
- **D.** Créer des clusters

Réponse B — Un rôle de projet délègue des droits limités au projet, avec ses propres jetons.

Révéler la réponse

QCM d'ancrage (3/4)

Questions 5 à 6

5. Quelle mesure ferme l'exécution de commandes dans les pods depuis l'UI ?

- **A.** `admin.enabled: "false"`
- **B.** `exec.enabled: "false"`
- **C.** `users.anonymous.enabled: "false"`
- **D.** `policy.default: ""`

Réponse B — `exec.enabled: "false"` désactive le terminal web.

Révéler la réponse

6. Quel risque majeur faut-il couvrir avec Sealed Secrets ?

- **A.** Perte de la clé privée du contrôleur (secrets indéchiffrables)
- **B.** Lenteur de la synchronisation
- **C.** Taille des fichiers chiffrés
- **D.** Incompatibilité avec Kustomize

Réponse A — Sans la clé privée, aucun SealedSecret ne peut être déchiffré.

Révéler la réponse

QCM d'ancrage (4/4)

Questions 7 à 7

7. Quelle approche garde la valeur du secret HORS de Git ?

- **A.** Sealed Secrets
- **B.** SOPS
- **C.** External Secrets Operator
- **D.** ConfigMap

Réponse C — External Secrets Operator lit la valeur dans un coffre ; Git ne contient qu'une référence.

Révéler la réponse

🎓 Module 12 : Meilleures pratiques et scénarios réels

Durée

1 h 30 (S12.1 : 35 min · S12.2 : 40 min · S12.3 : 15 min)

Prérequis

M00 à M11

Fil rouge

Hexa Mobilité passe de trois environnements à une plateforme qui grandit

Objectifs

- Planifier la mise à niveau et la sauvegarde d'une instance ArgoCD
- Générer des Applications en série avec un ApplicationSet
- Analyser une étude de cas et y appliquer les bonnes pratiques
- Situer les évolutions récentes de GitOps et d'ArgoCD

Déroulé

- Séquence S12.1 — Maintenance et mise à l'échelle (35 min)
- Séquence S12.2 — Études de cas et projet fictif (40 min)
- Séquence S12.3 — Tendances, synthèse, évaluation (15 min)

Séquence S12.1

Maintenance et mise à l'échelle

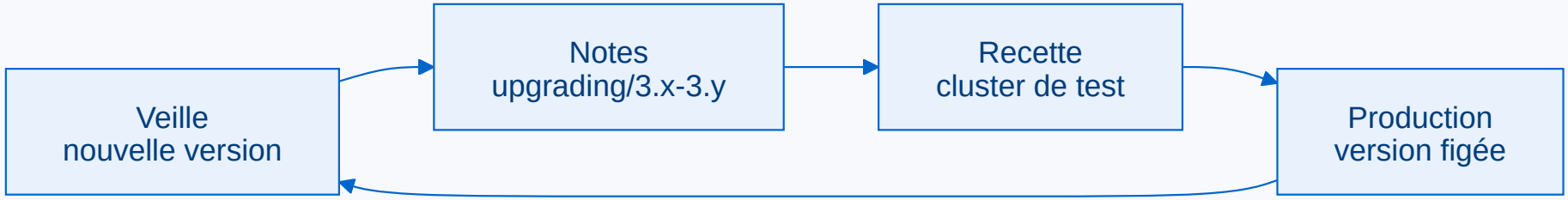
35 min

1. Faire vivre ArgoCD
2. Mettre à niveau
3. Sauvegarder et restaurer
4. Passer à l'échelle
5. ApplicationSet : une fabrique d'Applications
6. ApplicationSet en pratique
7. App of Apps et ArgoCD géré par ArgoCD

🕮 Faire vivre ArgoCD

Une version mineure par trimestre, trois versions mineures maintenues

`Maintenance = Veille + Notes de mise à niveau + Recette + Production`



- Correctifs uniquement pour les 3 dernières versions mineures (3.6, 3.5, 3.4 au 08/11/2026)
- Une version mineure à la fois ; notes `upgrading/` lues pour chaque saut

🕮 Mettre à niveau

Le même geste que l'installation, avec la nouvelle version

```bash
kubectl apply -n argocd --server-side --force-conflicts \
  -f https://raw.githubusercontent.com/argoproj/argo-cd/<NOUVELLE_VERSION>/manifests/install.yaml
kubectl rollout status statefulset/argocd-application-controller -n argocd
argocd version
```

- Manifeste complet réappliqué (recommandation officielle, même si seule l'image change)
- Personnalisations en Kustomize (défi 02.4) : une seule ligne modifiée, revue en merge request
- ArgoCD géré par ArgoCD (*self-managed*) : la mise à niveau devient un commit

⚠ Piège courant

saut de plusieurs versions mineures sans lire chaque note de mise à niveau

🕮 Sauvegarder et restaurer

Git contient l'essentiel… mais pas tout

```bash
argocd admin export -n argocd > argocd-backup.yaml
grep -E '^kind:' argocd-backup.yaml | sort | uniq -c
```

Sortie observée :

```
      3 kind: Application
      1 kind: ApplicationSet
      2 kind: AppProject
      4 kind: ConfigMap
      3 kind: Secret
```

- Restauration : `argocd admin import -n argocd - < argocd-backup.yaml`
- Fichier sensible (Secrets : dépôts, comptes, webhook) : stockage chiffré, hors Git
- Clé de Sealed Secrets à sauvegarder séparément (M11)

🕮 Passer à l'échelle

Quatre axes de croissance, quatre réponses

CROISSANCE	RÉPONSE	RÉGLAGE
Nombre d'applications	ApplicationSet	Générateurs (liste, Git, clusters)
Nombre de clusters	Multi-clusters, Argo CD Agent	`argocd cluster add`, agent par cluster
Charge du contrôleur	Sharding (StatefulSet, N réplicas)	`ARGOCD_CONTROLLER_REPLICAS`, `controller.sharding.algorithm`
Charge de rendu	Réplicas du `repo-server`	`reposerver.parallelism.limit`

- Disponibilité : manifeste HA (réplicas, Redis HA, 3 nœuds minimum, M02)

🕮 ApplicationSet : une fabrique d'Applications

Un dossier d'overlay = une Application

```yaml
spec:
  goTemplate: true
  generators:
    - git:
        repoURL: https://gitlab.com/<GITLAB_USER>/flotte-config.git
        revision: main
        directories:
          - path: apps/flotte-web/overlays/*
          - {path: apps/flotte-web/overlays/prod, exclude: true}
  template:
    metadata: {name: 'flotte-web-{{.path.basename}}'}
    spec:
      source: {path: '{{.path.path}}'}
      destination: {namespace: 'flotte-{{.path.basename}}'}
```

ApplicationSet en pratique

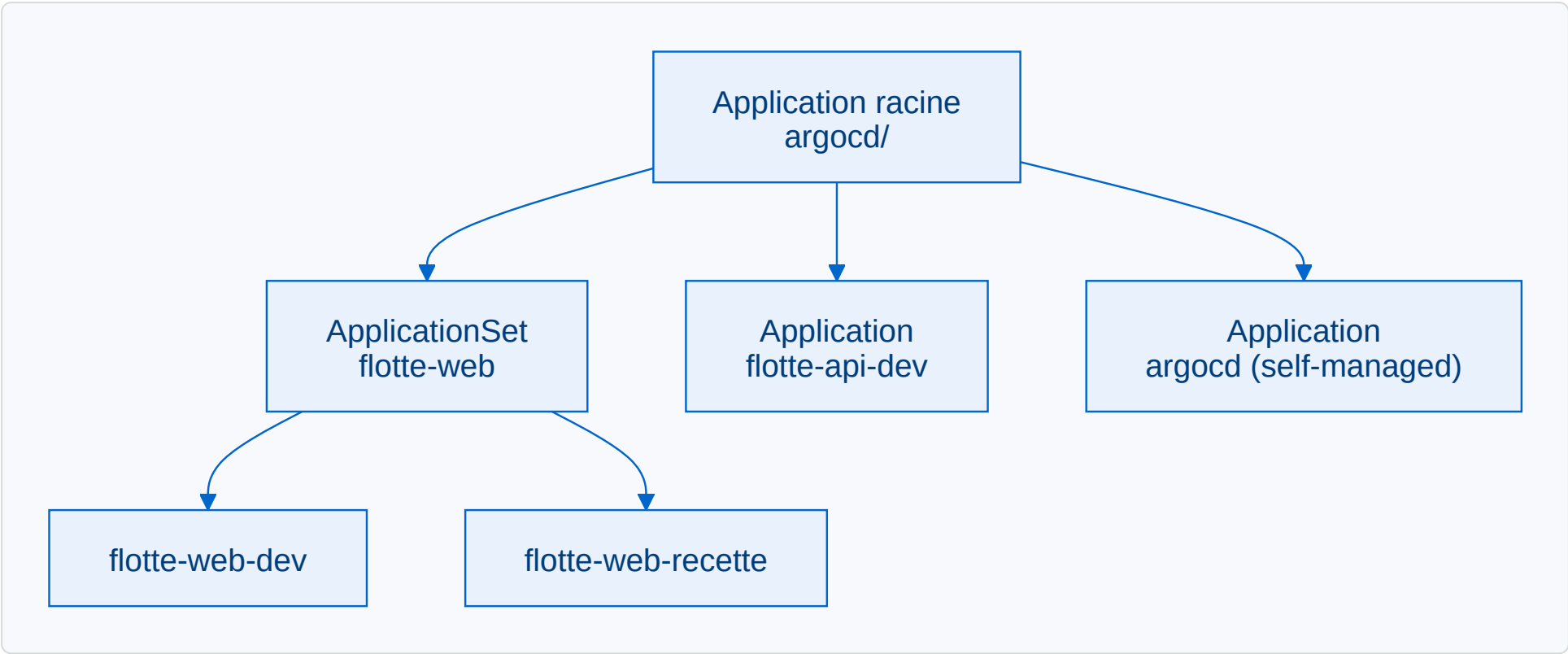
Comportement testé sur `flotte-web`

```
NAME                       NAMESPACE      STATUS  HEALTH
argocd/flotte-web-demo     flotte-demo    Synced  Healthy    <- dossier demo ajouté
argocd/flotte-web-dev      flotte-dev     Synced  Healthy
argocd/flotte-web-recette  flotte-recette Synced  Healthy
```

- Applications existantes de même nom : adoptées (propriétaire = ApplicationSet)
- Dossier supprimé → Application ET ressources supprimées (namespace conservé)
- Protection : `syncPolicy.applicationsSync: create-update` (option du contrôleur à activer)
- Bascule progressive entre Applications générées : *Progressive Syncs* (bêta depuis 3.3)

🕮 App of Apps et ArgoCD géré par ArgoCD

Une Application racine qui crée les autres



- Reconstruction d'une plateforme complète : installer ArgoCD + créer la racine

Séquence S12.2

Études de cas et projet fictif

40 min

1. Qui utilise ArgoCD ?
2. Cas type 1 : plateforme mutualisée
3. Cas type 2 : multi-clusters, multi-régions
4. Cas type 3 : environnement réglementé
5. Grille d'analyse d'un cas
6. Projet fictif : bilan *Hexa Mobilité*

Qui utilise ArgoCD ?

Un projet CNCF diplômé, largement adopté

- 448 organisations déclarées dans `USERS.md` du dépôt Argo CD (v3.6.1)
- Exemples déclarés : Adobe, Intuit (à l'origine d'Argo CD), Red Hat, Swisscom, Ticketmaster, Back Market
- Distribution commerciale : Red Hat OpenShift GitOps (basé sur Argo CD)
- Projet CNCF au niveau *graduated* (2022)

Source

liste déclarative tenue par le projet, sans détail des usages

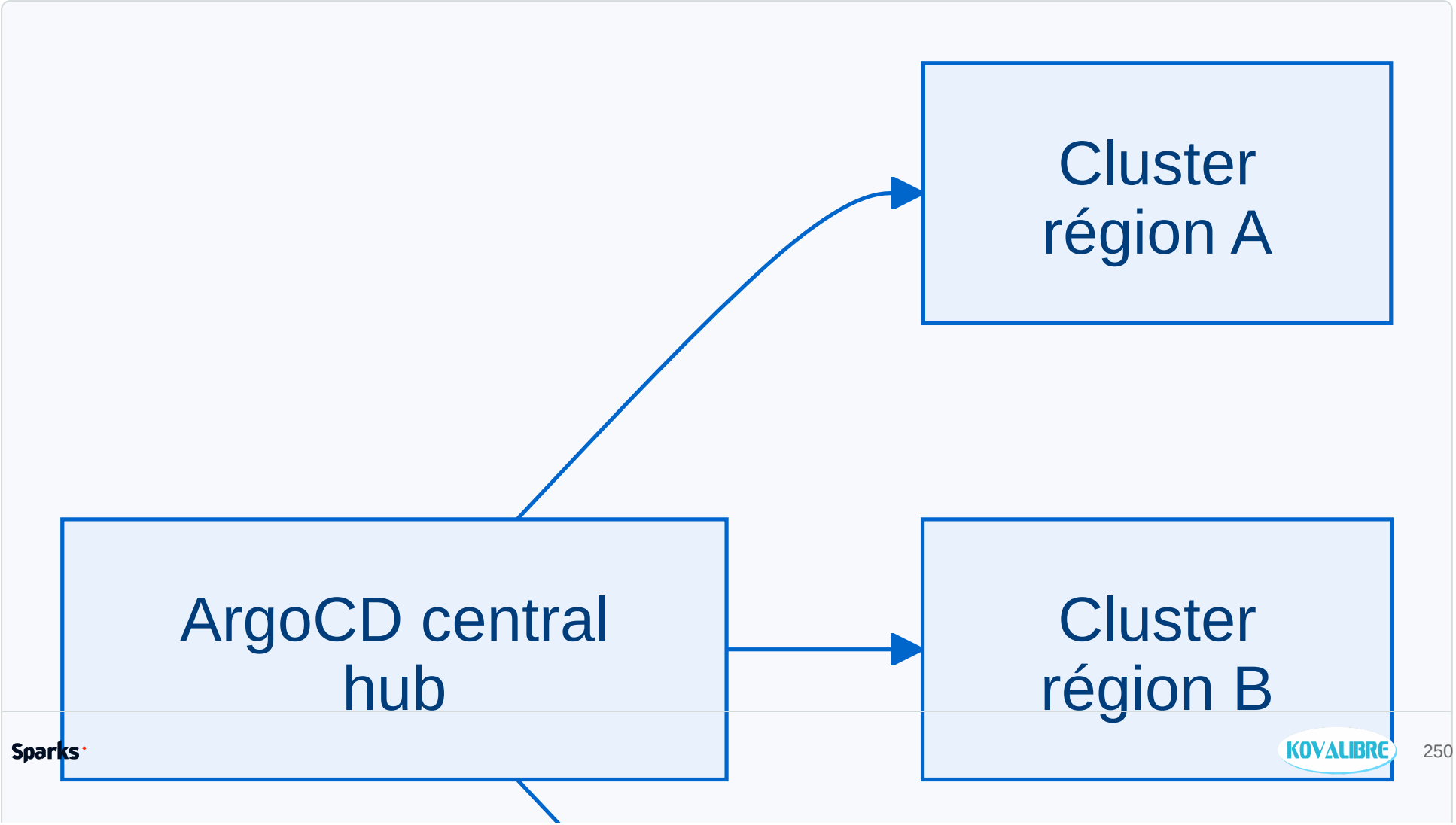
🕮 Cas type 1 : plateforme mutualisée

Une instance ArgoCD, vingt équipes produit

ENJEU	PRATIQUE
Isolation des équipes	Un AppProject par équipe, destinations `<equipe>-*`
Identité	SSO, groupes de l'annuaire → rôles (M11)
Création des environnements	ApplicationSet par équipe (générateur Git)
Autonomie sans risque	Rôles de projet, jetons CI propres à l'équipe
Charge	Sharding du contrôleur, réplicas du `repo-server`

🕮 Cas type 2 : multi-clusters, multi-régions

Plusieurs clusters de production, un pilotage central



🕮 Cas type 3 : environnement réglementé

Banque, santé, secteur public : chaque changement doit être prouvé

EXIGENCE	PRATIQUE ARGOCD
Changement approuvé	Merge request obligatoire sur `flotte-config`
Créneaux de mise en production	Fenêtres de synchronisation (`syncWindows` du projet)
Production décidée	Synchronisation manuelle, tag Git (M09)
Traçabilité	Historique Git + historique ArgoCD + notifications
Secrets	Coffre externe (External Secrets Operator)

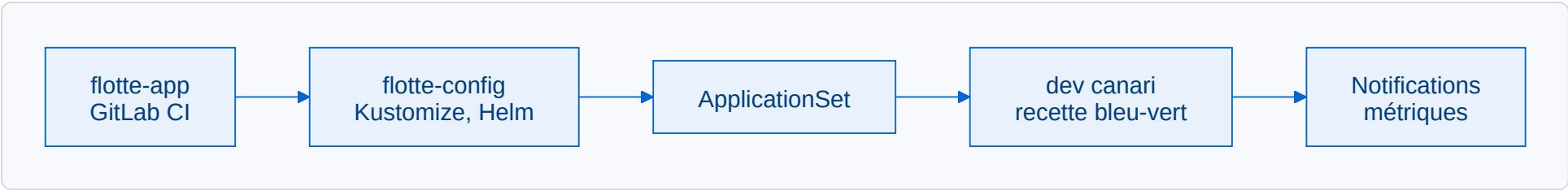
🕮 Grille d'analyse d'un cas

Les questions à poser avant de concevoir

THÈME	QUESTIONS
Organisation	Combien d'équipes, de dépôts, de clusters ? Qui valide ?
Sécurité	Annuaire ? Coffre ? Exigences d'audit ?
Livraison	Fréquence ? Canari, bleu-vert ? Fenêtres imposées ?
Exploitation	Qui surveille ? Quelles alertes ? Quelle reprise ?
Croissance	Nombre d'applications dans 1 an ? Nouveaux clusters ?

🕮 Projet fictif : bilan *Hexa Mobilité*

Du `kubectl apply` manuel à la plateforme GitOps



- Sécurité : projet `flotte`, RBAC, rôles de projet, Sealed Secrets
- Réversibilité : `git revert`, `app rollback`, `abort` du canari

Séquence S12.3

Tendances, synthèse, évaluation

15 min

1. Où va GitOps avec ArgoCD ?
2. Synthèse de la formation

❓ Où va GitOps avec ArgoCD ?

Évolutions documentées dans les versions récentes

ÉVOLUTION	STATUT (V3.6.1)	APPORT
Source Hydrator	Bêta depuis 3.5	Manifestes rendus écrits dans Git (relecture du YAML final)
Sources OCI	Documentées (`oci.md`)	Configuration publiée comme artefact OCI
Progressive Syncs	Bêta depuis 3.3	Déploiement ordonné des Applications d'un ApplicationSet
Argo CD Agent	Projet `argoproj-labs` , v0.10.0	Pilotage de clusters non joignables

- Fil commun : plus de clusters, plus d'automatisation, plus de traçabilité

📌 Synthèse de la formation

Les 6 parties du programme

PARTIE	ACQUIS
1. Introduction	GitOps, architecture, composants
2. Configuration	Installation, accès, RBAC, projets, dépôts
3. Déploiement	Applications, synchronisation, hooks, rollback
4. Gestion	Supervision, dépannage, bleu-vert, canari
5. Avancé	Helm, Kustomize, CI, sécurité, secrets
6. Bonnes pratiques	Maintenance, échelle, cas réels, tendances

- Évaluation finale : `evaluation-finale.md` (QCM et cas pratique)

À retenir

Points clés

- Mise à niveau : version par version, notes lues, manifeste complet réappliqué
- Sauvegarde : `argocd admin export` (fichier sensible) + clé de Sealed Secrets
- ApplicationSet : un générateur, N Applications ; attention à la suppression
- Études de cas : organisation, sécurité, livraison, exploitation, croissance
- Tendances : rendu dans Git, OCI, synchronisations progressives, agents

QCM d'ancrage (1/4)

Questions 1 à 2

1. Combien de versions mineures d'ArgoCD reçoivent des correctifs ?

- **A.** Une seule, la dernière
- **B.** Les trois plus récentes
- **C.** Toutes les versions 3.x
- **D.** Les cinq plus récentes

Réponse B — Seules les trois versions mineures les plus récentes reçoivent des correctifs.

Révéler la réponse

2. Que contient la sauvegarde `argocd admin export` ?

- **A.** Uniquement les Applications
- **B.** Applications, projets, ApplicationSets, ConfigMaps et Secrets d'ArgoCD
- **C.** Les ressources déployées dans les clusters
- **D.** Le contenu des dépôts Git

Réponse B — Applications, projets, ApplicationSets, ConfigMaps et Secrets d'ArgoCD.

Révéler la réponse

QCM d'ancrage (2/4)

Questions 3 à 4

3. Avec le générateur Git `directories`, que se passe-t-il quand un dossier d'overlay est supprimé ?

- **A.** Rien
- **B.** L'Application générée et ses ressources sont supprimées
- **C.** L'ApplicationSet passe en erreur
- **D.** Le dossier est recréé

Réponse B — L'Application générée disparaît avec ses ressources (politique `sync` par défaut).

Révéler la réponse

4. Quelle réponse à la croissance du nombre de clusters non joignables depuis un hub central ?

- **A.** Augmenter les réplicas du `repo-server`
- **B.** Un agent par cluster (Argo CD Agent)
- **C.** Désactiver la synchronisation automatique
- **D.** Un AppProject par cluster

Réponse B — Un agent par cluster (Argo CD Agent) quand le hub ne peut pas joindre les clusters.

Révéler la réponse

QCM d'ancrage (3/4)

Questions 5 à 6

5. Quel mécanisme impose des créneaux de mise en production ?

- **A.** Les hooks PreSync
- **B.** Les fenêtres de synchronisation du projet
- **C.** Les sync waves
- **D.** Le RBAC

Réponse B — Les fenêtres de synchronisation (`syncWindows`) de l'AppProject.

Révéler la réponse

6. Que fait le Source Hydrator (bêta depuis 3.5) ?

- **A.** Il chiffre les secrets
- **B.** Il écrit dans Git les manifestes rendus
- **C.** Il remplace Helm
- **D.** Il supprime les ressources orphelines

Réponse B — Il écrit dans Git les manifestes rendus.

Révéler la réponse

QCM d'ancrage (4/4)

Questions 7 à 7

7. Pourquoi conserver les personnalisations d'ArgoCD dans un `kustomization.yaml` ?

A. Pour accélérer ArgoCD

B. Pour qu'une mise à niveau se résume à changer la version de la ressource distante

C. Parce que le manifeste officiel est en lecture seule

D. Pour activer la haute disponibilité

Réponse B — La mise à niveau se résume à changer l'URL de version dans le `kustomization.yaml`.

Révéler la réponse