DEVOPS & DÉPLOIEMENT

Docker pour développeurs web

Conteneuriser une application Node.js, PHP ou Python de zéro. Comprendre les images, les conteneurs, les volumes et Docker Compose : le point d'entrée DevOps le plus demandé par les développeurs juniors et les équipes qui veulent en finir avec le "ça marche sur ma machine".

DURÉE	TARIF HT	NIVEAU	LANGUE	GROUPE	FORMAT
3j (21h)	~~900.00 €~~ **540.00 €**	**Débutant**	**Français**	**1-6**	**Formation**

1 PUBLIC VISÉ

- Développeurs web juniors ou intermédiaires (Node.js, PHP, Python) souhaitant comprendre et adopter Docker dans leur workflow quotidien.
- Développeurs travaillant en équipe qui veulent harmoniser les environnements de développement et en finir avec les problèmes de configuration locale.
- Freelances souhaitant livrer des applications conteneurisées et maîtriser le déploiement sur un VPS ou une plateforme cloud.
- Développeurs backend ayant suivi les formations FastAPI, Symfony ou Spring Boot et souhaitant conteneuriser leurs projets.
- Profils techniques curieux du DevOps cherchant un point d'entrée concret et immédiatement applicable.

2 PRÉREQUIS

- Maîtrise du terminal Linux/macOS : navigation dans les répertoires, manipulation de fichiers, variables d'environnement, redirections.
- Expérience avec au moins un langage web côté serveur : Node.js, PHP ou Python — savoir lancer une application localement.
- Notions de base sur les formats JSON et YAML.
- Avoir déjà consommé une API REST ou interagi avec une base de données depuis du code.
- Aucune connaissance de Docker, Kubernetes ou d'infrastructure cloud requise.

3 OBJECTIFS PÉDAGOGIQUES

- Comprendre l'architecture Docker : daemon, client, images, conteneurs, registres
- Écrire des Dockerfiles efficaces pour des applications Node.js, PHP et Python
- Maîtriser les commandes Docker essentielles : build, run, exec, logs, inspect, prune
- Gérer la persistance des données avec les volumes et les bind mounts
- Orchestrer plusieurs services avec Docker Compose : app, base de données, cache
- Optimiser les images : multi-stage builds, cache des layers, réduction de la taille
- Configurer les réseaux Docker et la communication inter-conteneurs
- Déployer une stack conteneurisée sur un VPS avec Docker Compose en production

4 PROGRAMME DÉTAILLÉ

Jour 1 (7h) Fondamentaux Docker et premiers conteneurs

Module 1 Comprendre Docker : architecture et concepts clés (2h)

Pourquoi Docker, et comment ça fonctionne (1h)

- Le problème que Docker résout : "ça marche sur ma machine", onboarding laborieux, conflits de dépendances
- Conteneurs vs machines virtuelles : isolation légère, partage du kernel, démarrage instantané
- Les composants Docker : daemon (dockerd), CLI (docker), registre (Docker Hub)
- Images vs conteneurs : l'image est la recette, le conteneur est l'instance en cours d'exécution
- Les layers d'une image : système de fichiers en couches, cache, réutilisation
- Le cycle de vie d'un conteneur : created → running → stopped → removed

Cas pratique : installer Docker Desktop, lancer docker run hello-world, inspecter les layers de l'image avec docker history

Les commandes Docker essentielles (1h)

- Gérer les images : pull, images, rmi, inspect
- Gérer les conteneurs : run, ps, stop, rm, exec, logs
- Les options clés de docker run : -d, -p, -e, --name, --rm
- Entrer dans un conteneur en cours d'exécution : docker exec -it
- Inspecter et déboguer : docker inspect, docker stats, docker top
- Nettoyer les ressources inutilisées : docker system prune, docker volume prune

Cas pratique : lancer un conteneur Nginx, mapper le port 8080, modifier une page HTML depuis l'hôte, observer les logs en temps réel

Module 2 Écrire des Dockerfiles (3h)

Anatomie d'un Dockerfile (1h)

- Les instructions fondamentales : FROM, WORKDIR, COPY, RUN, CMD, EXPOSE
- Choisir une image de base : debian vs alpine vs distroless — taille, sécurité, compatibilité
- CMD vs ENTRYPOINT : quand utiliser l'un, l'autre, ou les deux
- Les variables d'environnement dans le Dockerfile : ENV vs ARG
- Le fichier .dockerignore : exclure node_modules, .env, .git
- Construire et tagger une image : docker build -t monapp:1.0 .

Cas pratique : Dockerfile pour une application "Hello World" en Python (Flask) — construire, lancer, accéder dans le navigateur

Dockeriser une application Node.js (1h)

- Image de base node:20-alpine : pourquoi alpine, taille comparée
- Installer les dépendances : copier package.json avant le code source pour profiter du cache
- L'ordre des instructions et le cache des layers : l'optimisation la plus simple et la plus impactante
- Lancer l'app avec node server.js ou npm start — différences
- Gestion des signaux : utiliser dumb-init ou tini comme PID 1

Cas pratique : conteneuriser une API Express existante — Dockerfile optimisé, image < 150 Mo, démarrage < 2 secondes

Dockeriser une application PHP et Python (1h)

- PHP : image php:8.3-fpm-alpine, installer les extensions avec docker-php-ext-install
- PHP : gérer les dépendances Composer dans le build sans installer Composer dans l'image finale
- Python : image python:3.12-slim, installer les dépendances avec requirements.txt
- Python : FastAPI ou Django — adapter le CMD pour uvicorn ou gunicorn
- Bonnes pratiques communes : utilisateur non-root, répertoire de travail explicite, pas de secrets dans l'image

Cas pratique : chaque participant conteneurise l'application du langage de son choix — revue collective des Dockerfiles et identification des optimisations

Module 3 Volumes et persistance des données (2h)

Volumes Docker et bind mounts (1h)

- Le problème de la persistance : pourquoi les données disparaissent quand un conteneur est supprimé
- Les volumes nommés (docker volume create) : gérés par Docker, portables
- Les bind mounts : monter un répertoire local dans le conteneur — parfait pour le développement
- Les tmpfs mounts : données en mémoire, jamais écrites sur disque
- Inspecter et gérer les volumes : docker volume ls, inspect, prune

Cas pratique : lancer un conteneur MySQL avec un volume nommé — arrêter, supprimer et relancer le conteneur, vérifier que les données persistent

Live reload en développement avec les bind mounts (1h)

- Monter le code source depuis l'hôte pour développer sans reconstruire l'image
- Node.js : nodemon dans le conteneur pour redémarrer sur chaque modification
- Python : uvicorn --reload ou flask --debug dans le conteneur
- PHP : pas de redémarrage nécessaire — bind mount suffit avec PHP-FPM
- Performances des bind mounts sur macOS et Windows : delegated, cached, solutions alternatives

Cas pratique : workflow de développement complet — modifier un fichier source sur l'hôte, voir le changement reflété immédiatement dans le conteneur sans rebuild

Jour 2 (7h) Docker Compose, réseaux et optimisation des images

Module 4 Docker Compose : orchestrer une stack complète (4h)

Introduction à Docker Compose (1h)

- Pourquoi Compose : gérer plusieurs conteneurs liés avec un seul fichier YAML
- Structure du fichier compose.yaml : services, networks, volumes
- Les commandes Compose essentielles : up, down, ps, logs, exec, build
- Différence entre image et build dans un service
- Les variables d'environnement dans Compose : environment, env_file, interpolation depuis .env

Ordre de démarrage et dépendances : depends_on et healthcheck

Cas pratique : stack à deux services — application web + base de données PostgreSQL — démarrée avec docker compose up -d

Composer une stack Node.js + PostgreSQL + Redis (1h30)

- Déclarer le service app avec build context et variables d'environnement
- Déclarer le service db (PostgreSQL) avec volume nommé pour la persistance
- Déclarer le service cache (Redis) avec configuration personnalisée
- Connecter l'application à la base de données via le nom du service comme hostname
- Les healthchecks : attendre que PostgreSQL soit prêt avant de démarrer l'app
- Profiles Compose : activer/désactiver des services selon l'environnement

Cas pratique : API Express connectée à PostgreSQL et Redis via Compose — connexion via pg et ioredis, données persistées, cache fonctionnel

Stack PHP + Nginx + MySQL et stack Python + FastAPI + PostgreSQL (1h30)

- PHP-FPM + Nginx : deux conteneurs qui coopèrent — Nginx sert les fichiers statiques et proxifie vers PHP-FPM
- Configurer Nginx dans le conteneur : fastcgi_pass vers le service PHP par nom de service
- MySQL dans Compose : initialisation de la base avec un script SQL au premier démarrage
- FastAPI + PostgreSQL : attendre que la base soit prête avec un healthcheck, migrations Alembic au démarrage
- Override Compose : compose.override.yaml pour les différences dev vs prod

Cas pratique : atelier au choix — stack PHP/Nginx/MySQL ou stack FastAPI/PostgreSQL — Compose fonctionnel, données persistées, rebuild automatique en dev

Module 5 Réseaux Docker (1h30)

Comprendre les réseaux Docker (45min)

- Les drivers réseau : bridge, host, none — quand utiliser lequel
- Le réseau bridge par défaut vs les réseaux bridge nommés : isolation et DNS interne
- La résolution DNS dans Docker : les conteneurs se trouvent par leur nom de service
- Inspecter un réseau : docker network inspect, adresses IP, passerelles
- Créer des réseaux nommés dans Compose : isoler les services front-end et back-end

Exposition des ports et sécurité réseau (45min)

- Mapper les ports : -p 8080:80 — port hôte vs port conteneur
- Exposer uniquement les services nécessaires vers l'hôte : ne pas exposer la base de données
- Communication inter-conteneurs sans passer par l'hôte : utiliser le réseau interne Compose
- Reverse proxy avec Nginx ou Traefik : un seul point d'entrée, plusieurs services derrière

Cas pratique : stack avec deux réseaux Compose — réseau frontend (app + Nginx) et réseau backend (app + base) — la base n'est pas accessible depuis l'extérieur

Module 6 Optimiser les images : multi-stage builds (1h30)

Le problème des images lourdes (30min)

- Pourquoi la taille de l'image compte : temps de pull, surface d'attaque, stockage registre
- Ce qui alourdit une image : outils de build, dépendances de dev, cache package manager
- Analyser la taille d'une image layer par layer avec docker history et dive

Multi-stage builds (1h)

- Le principe : plusieurs étapes FROM dans un même Dockerfile, copier seulement les artefacts nécessaires
- Stage "builder" : installer les outils de build, compiler, bundler
- Stage "runner" : image minimale, copier uniquement le binaire ou les fichiers de prod
- Multi-stage pour Node.js : builder avec npm, runner avec node:alpine sans npm ni devDependencies
- Multi-stage pour Python : builder avec pip, runner avec image slim sans pip ni gcc
- Multi-stage pour PHP : builder avec Composer, runner sans Composer
- Résultats concrets : passer de 800 Mo à moins de 100 Mo sur une app Node.js typique

Cas pratique : transformer le Dockerfile du Jour 1 en multi-stage — mesurer la réduction de taille avant/après avec docker images

Jour 3 (7h) Déploiement, CI/CD et projet de synthèse

Module 7 Registres et publication d'images (1h)

Docker Hub et registres privés (1h)

- Tagger une image pour la publication : convention username/image:tag
- Pousser et puller depuis Docker Hub : docker push, docker pull
- Les registres alternatifs : GitHub Container Registry (ghcr.io), GitLab Registry, AWS ECR
- Authentification dans un pipeline CI : variables d'environnement, secrets GitHub Actions
- Versionner les images : tags sémantiques, tag latest, tags de commit Git

Cas pratique : tagger et pousser l'image construite en Jour 1 sur Docker Hub — la puller sur une autre machine et vérifier qu'elle fonctionne

Module 8 Déploiement sur un VPS avec Docker Compose (2h)

Préparer l'environnement de production (1h)

- Installer Docker sur un VPS Ubuntu : script officiel, post-installation, groupe docker
- Différences entre Compose dev et Compose prod : pas de bind mount, pas de rebuild, restart policy
- Les restart policies : always, unless-stopped, on-failure
- Gérer les secrets en production : variables d'environnement depuis un fichier .env non commité, Docker Secrets
- Nginx comme reverse proxy devant l'application : configuration proxy_pass
- HTTPS avec Certbot et Let's Encrypt : certificat SSL automatique

Déployer et mettre à jour sans interruption (1h)

- Workflow de déploiement manuel : git pull → docker compose pull → docker compose up -d
- Les rolling updates avec Compose : docker compose up -d --no-deps --build app

Sauvegarder et restaurer un volume de base de données : docker exec + pg_dump / mysqldump

- Surveiller les conteneurs en production : docker stats, docker compose logs -f

Cas pratique : déployer la stack Compose sur un VPS (fourni ou simulé) — application accessible en HTTPS, base de données persistée, restart automatique

Module 9 Intégration dans un pipeline CI/CD avec GitHub Actions (1h)

Automatiser le build et le push d'image (1h)

- Workflow GitHub Actions : déclencher sur push sur main
- Étapes : checkout → login Docker Hub → build → push avec tag du commit SHA
- Le cache de layers dans GitHub Actions : cache-from et cache-to avec gha
- Déploiement automatique sur le VPS : SSH depuis GitHub Actions avec une clé déployée
- Bonnes pratiques : ne jamais stocker de secrets dans le code, utiliser GitHub Secrets

Cas pratique : pipeline GitHub Actions complet — push sur main → build → push ghcr.io → déploiement automatique sur le VPS

Module 10 Projet de synthèse (3h)

Conteneuriser et déployer une application complète de bout en bout

- **Cahier des charges :** conteneuriser une application existante (apportée par le participant ou fournie) incluant au moins un service applicatif et une base de données
- **Livrables attendus :** Dockerfile multi-stage optimisé, fichier compose.yaml avec au moins deux services, réseaux Docker correctement configurés, volume pour la persistance des données, fichier compose.prod.yaml prêt pour la production
- Étape 1 : Dockerfile et build de l'image — taille cible en dessous de 150 Mo
- Étape 2 : Docker Compose avec service applicatif + base de données + healthcheck
- Étape 3 : configuration production (restart policy, variables d'environnement, Nginx)
- Étape 4 : déploiement sur VPS ou présentation de la stack en local
- Revue collective : choix d'images de base, stratégie de layers, sécurité des secrets
- Retour formateur individualisé sur le Dockerfile et le Compose rendu

5 COMPÉTENCES VISÉES

- Écrire des Dockerfiles optimisés avec multi-stage build pour Node.js, PHP et Python
- Orchestrer une stack complète (app + base de données + cache) avec Docker Compose
- Gérer la persistance des données avec les volumes et les bind mounts
- Configurer les réseaux Docker pour isoler et connecter les services
- Déployer une stack conteneurisée sur un VPS avec HTTPS et restart automatique
- Automatiser le build et le déploiement via un pipeline GitHub Actions

6 MODALITÉS PÉDAGOGIQUES

Formation délivrée en présentiel ou distanciel. Le formateur alterne entre méthode démonstrative (live coding dans le terminal avec des applications réelles en Node.js, PHP et Python), méthode interrogative (analyse des erreurs Docker fréquentes et discussion des choix d'architecture) et méthode active (exercices de conteneurisation et projet de synthèse fil rouge). L'accent est mis sur la maîtrise du terminal et la compréhension de ce qui se passe réellement dans les conteneurs.

7 MOYENS PÉDAGOGIQUES ET TECHNIQUES

- Support de cours numérique mis à disposition
- Dockerfiles et fichiers Compose annotés avec exercices et corrections par module
- Environnement : Docker Desktop + VS Code + accès à un VPS Ubuntu pour le déploiement
- Pour le distanciel : visioconférence, partage d'écran, chat en direct
- Accès à la plateforme pédagogique LaPolaris (supports, ressources, émargement)

8 MODALITÉS D'ÉVALUATION

- En cours de formation : exercices de conteneurisation corrigés et comparés à chaque module
- En fin de formation : réalisation d'une stack Docker Compose complète avec déploiement
- Questionnaire d'auto-évaluation des acquis en fin de parcours

9 CRITÈRES D'ÉVALUATION

- Dockerfile fonctionnel avec multi-stage build, image finale sous 150 Mo et utilisateur non-root
- Fichier compose.yaml opérationnel avec au moins deux services, healthcheck et volume persistant
- Réseaux Docker correctement configurés : base de données non exposée vers l'extérieur
- Bind mount fonctionnel en développement avec live reload sans rebuild de l'image
- Stack déployée sur VPS avec restart policy, variables d'environnement sécurisées et HTTPS
- Pipeline GitHub Actions opérationnel buildant et poussant l'image sur un registre

10 MODALITÉS DE VALIDATION

Attestation de fin de formation délivrée à l'issue du parcours, conditionnée à une assiduité d'au moins 80 % et à la réalisation du projet de synthèse. L'attestation précise les objectifs atteints et les compétences acquises.

11 SUIVI ET ACCOMPAGNEMENT

- Feuilles d'émargement signées par demi-journée (présentiel) ou émargement numérique (distanciel)
- Traçabilité des activités pédagogiques réalisées
- Attestation d'assiduité délivrée en fin de formation
- Suivi individuel via la vérification du dépôt GitHub de chaque apprenant en fin de parcours

12 CONDITIONS D'ACCÈS

Formation accessible sur inscription directe, sans prérequis administratif particulier. Le financement peut être pris en charge par l'employeur dans le cadre d'un plan de développement des compétences, ou en autofinancement avec possibilité de paiement en plusieurs fois.

13 DÉLAIS D'ACCÈS

Inscription possible jusqu'à **5 jours ouvrés** avant le début de la session. Pour toute demande urgente, nous contacter directement.

ACCESSIBILITÉ · HANDICAP

Nos formations sont accessibles aux personnes en situation de handicap. Pour toute situation nécessitant un aménagement (matériel, temporel ou pédagogique), nous vous invitons à nous contacter avant l'inscription afin d'étudier les adaptations possibles. Référent handicap : contact@lapolaris.fr

LaPolaris

TÉL. +33762584798

EMAIL contact@lapolaris.fr

WEB lapolaris.fr