



## Bloc 2

Doussain Jimmy

Expert en Développement Logiciel (RNCP 39583)



# Sommaire

<b>Sommaire</b>	<b>1</b>
<b>Introduction</b>	<b>3</b>
<b>2. Présentation du projet</b>	<b>4</b>
Fonctionnalités principales du MVP	4
<b>3. Déploiement continu</b>	<b>5</b>
Vérification après déploiement	5
<b>4. Intégration continue</b>	<b>6</b>
4.1 Étapes de la pipeline CI	6
4.2 Organisation avec GitFlow	6
<b>5. Architecture &amp; Prototype</b>	<b>7</b>
5.1 Vue d'ensemble	7
5.3 Cycle d'un run (end to end)	7
5.4 Prototype (écrans clés)	8
5.5 Modèle de données (résumé)	8
5.6 Décisions techniques marquantes	8
5.7 Sécurité & perf côté archi	8
<b>6. Tests unitaires</b>	<b>9</b>
6.1 Objectifs	9
6.2 Frontend — Vitest + React Testing Library	9
6.3 API — Japa + Supertest	9
6.4 Runner — Vitest (mocks dockerode)	9
<b>7. Sécurité &amp; Accessibilité</b>	<b>10</b>
7.1 Objectifs (court)	10
7.2 API (AdonisJS) — protections principales	10
7.3 Runner & Docker (points durs)	10
<b>8. Déploiement progressif</b>	<b>11</b>
8 Stratégies retenues (MVP)	11
<b>9. Cahier de recettes</b>	<b>12</b>
9.1 Préparation	12
9.2 Recette A — Authentification (login/logout)	12
9.3 Recette B — Création d'un projet + synchronisation YAML	12
9.4 Recette C — Lancer un run et suivre les logs	13
9.5 Recette D — Job de déploiement (simulation "command")	13
<b>10. Plan de correction des bogues (Redmine)</b>	<b>14</b>
<b>11. Manuel de déploiement</b>	<b>15</b>
<b>12. Manuel de mise à jour</b>	<b>17</b>
<b>13. Manuel d'utilisation</b>	<b>19</b>
<b>14. Choix techniques</b>	<b>22</b>
<b>15. Conclusion</b>	<b>24</b>

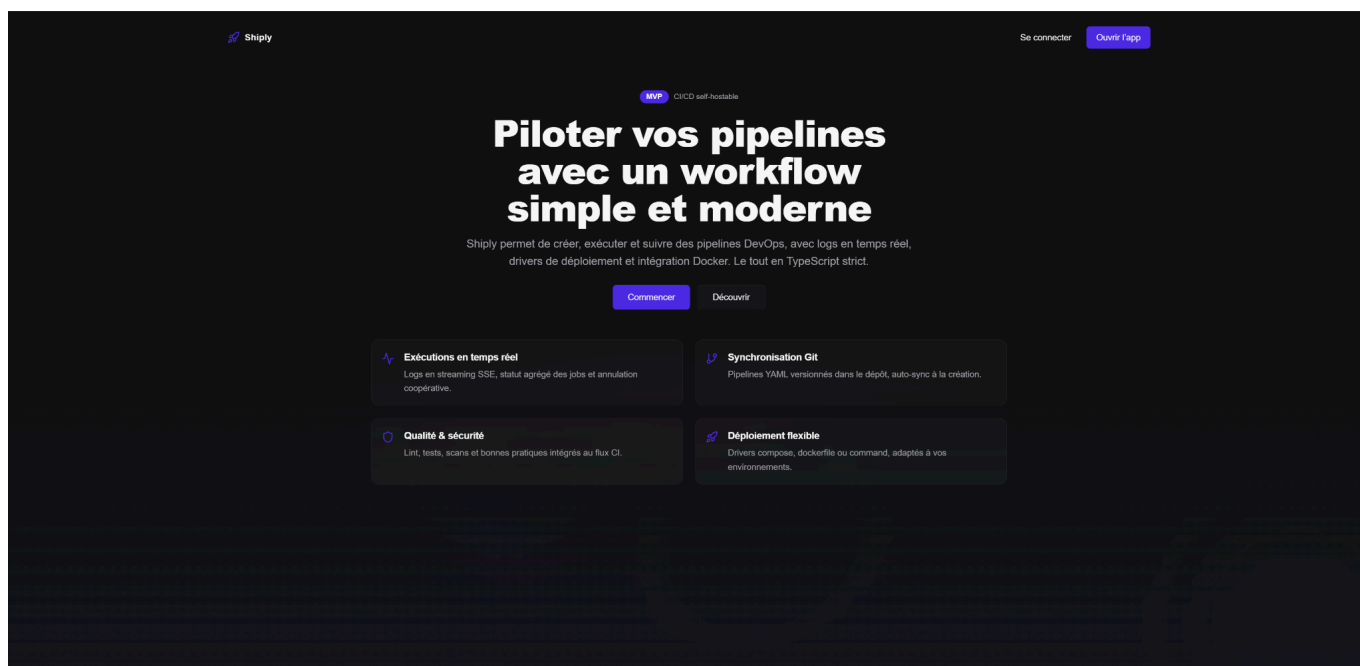
# Introduction.

Shiply, une plate-forme CI/CD auto-hébergeable développée dans le cadre d'un projet de fin d'année. L'objectif est de proposer une chaîne permettant de créer, exécuter et suivre des pipelines (build, tests, qualité, sécurité, déploiement) avec un runner Docker et un gestionnaire de déploiement pour serveur. Le projet met l'accent sur la lisibilité des statuts, le streaming des logs en temps réel, la reproductibilité des mises en production.

Shiply est construit comme un mono-repo **TypeScript** articulant trois briques principales : un **frontend** React (Vite, Tailwind, shadcn/ui) pour l'interface, une **API** AdonisJS (TypeScript, Lucid ORM, authentification par token) pour la logique et les données, et un **runner** Bun + dockerode pour l'exécution isolée des jobs dans des conteneurs.

Le mono-repo quant à lui est créé via TurboRepo : <https://turborepo.com/>

Le rapport est structuré de façon à couvrir l'ensemble des attendus C2.x. Après la présentation du projet, il détaille le protocole de déploiement continu ainsi que la pipeline d'intégration continue ainsi, il expose l'architecture et le prototype fonctionnel, décrit les tests, les mesures de sécurité et d'accessibilité, puis le déploiement continu et l'historique des versions.



## 2. Présentation du projet

---

Shiply est une plateforme CI/CD conçue pour répondre à mes besoins.

J'ai eu cette idée en cherchant une solution pour pouvoir déployer mes applications sur mon serveur. Il existe des outils comme Coolify, mais l'interface et le fonctionnement de l'outil ne s'accordait pas avec mes besoins.

Le projet repose sur un **mono-repo TypeScript** structuré autour de trois composants principaux

**Frontend (client/)** : une application React (Vite, TailwindCSS, shadcn/ui) offrant une interface accessible. Elle permet de gérer les projets, visualiser l'état des pipelines, suivre les logs en temps réel et contrôler les déploiements.

**API (server/)** : un serveur AdonisJS (TypeScript, Lucid ORM avec MySQL) qui assure la gestion des utilisateurs, des projets, des pipelines et des exécutions. L'authentification se fait par tokens JWT, et des événements SSE (Server-Sent Events) permettent le streaming des logs.

**Runner (runner/)** : un moteur d'exécution développé avec Bun et dockerode. Il orchestre les jobs dans des conteneurs éphémères, en respectant un espace de travail par projet. Le runner supporte également des drivers de déploiement (Compose, Dockerfile, Commande) pour automatiser la mise en production.

### 2.1 Fonctionnalités principales du MVP

Le MVP de Shiply regroupe un ensemble de fonctionnalités essentielles qui couvrent à la fois la gestion des projets, l'exécution des pipelines et le suivi en temps réel. Voici les points principaux :

- Création et gestion de projets reliés à des dépôts GitHub.
- Synchronisation et validation des pipelines définis en YAML.
- Exécution de runs composés de jobs conteneurisés, avec logs en direct.
- Déploiement automatisé via des drivers standardisés (docker compose, dockerfile, direct).
- Gestion des erreurs et possibilité d'annuler une exécution en cours.
- Accessibilité respectant les bonnes pratiques (navigation clavier, labels reliés, contrastes vérifiés).

### 3. Déploiement continu

---

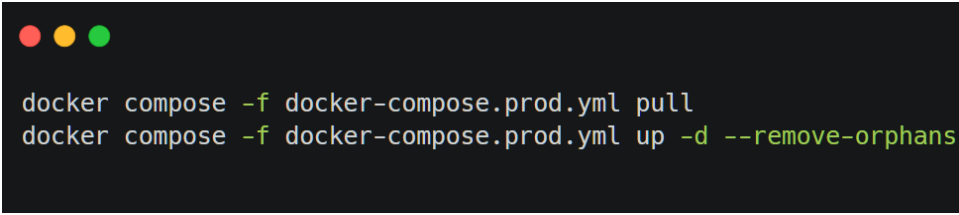
Le déploiement continu dans Shiply repose directement sur la pipeline CI/CD se trouvant à la racine dans :

- `.github/workflows/release.yml`

Lorsqu'une nouvelle version est push et taguée sur la branche `main` (par ex. `v1.2.0`), la pipeline se charge de tout:

1. **Build des images Docker** de chaque service (API, client, runner).
2. **Scan de sécurité avec Trivy** pour vérifier qu'il n'y a pas de failles connues.
3. **Push des images sur Docker Hub** avec un tag SemVer (`doxteurn/shiply-api:1.2.0`, etc.).
4. **Connexion automatique à mon serveur** (via SSH dans le workflow).
5. **Exécution des commandes Docker Compose** sur le serveur pour tirer les nouvelles images et redémarrer les services :

Les commandes exécutées en production par la CI/CD :

A screenshot of a terminal window with a dark background and three colored window control buttons (red, yellow, green) in the top left corner. The terminal displays two lines of commands: `docker compose -f docker-compose.prod.yml pull` and `docker compose -f docker-compose.prod.yml up -d --remove-orphans`.

```
docker compose -f docker-compose.prod.yml pull
docker compose -f docker-compose.prod.yml up -d --remove-orphans
```

#### Vérification après déploiement

Une fois le déploiement terminé, un endpoint `/health` permet de vérifier que tout fonctionne correctement.

Si jamais une mise à jour pose un problème, je peux redéployer une version précédente en utilisant simplement son tag Docker.

## 4. Intégration continue

---

L'intégration continue dans Shiply a pour but de vérifier automatiquement que tout le code reste de bonne qualité à chaque modification.

Dès qu'une nouvelle fonctionnalité est poussée ou qu'une pull request est ouverte, une pipeline GitHub Actions se déclenche et applique une série d'étapes obligatoires.

### 4.1 Étapes de la pipeline CI

1. **Installation des dépendances** → toutes les dépendances du monorepo sont installées avec Bun.
2. **Typecheck** → TypeScript vérifie que le code respecte les types partagés (**shared/**). Cela permet de détecter rapidement les erreurs entre le client, le serveur et le runner.
3. **Lint** → ESLint s'assure que le code suit les conventions de style et évite les mauvaises pratiques.
4. **Tests unitaires et d'intégration** →
  - côté **frontend**, j'utilise Vitest pour tester les composants et les pages,
  - côté **API**, j'utilise Japa + Supertest pour valider les endpoints et la logique des services.Un minimum de **80 % des modules touchés** doit être couvert.
5. **Build** → si les tests passent, la pipeline génère les builds optimisés (frontend, API, runner).
6. **Analyse de sécurité** → Trivy scanne les dépendances et les images Docker pour détecter les vulnérabilités.
7. **Audit performance** → sur le frontend, Lighthouse CI (LHCI) vérifie les performances, l'accessibilité et le SEO.

PS: [Le rapport Lighthouse est disponible en annexe.](#)

### 4.2 Organisation avec GitFlow

Pour éviter toute erreur en production sur la branche **main**, j'utilise un workflow GitFlow :

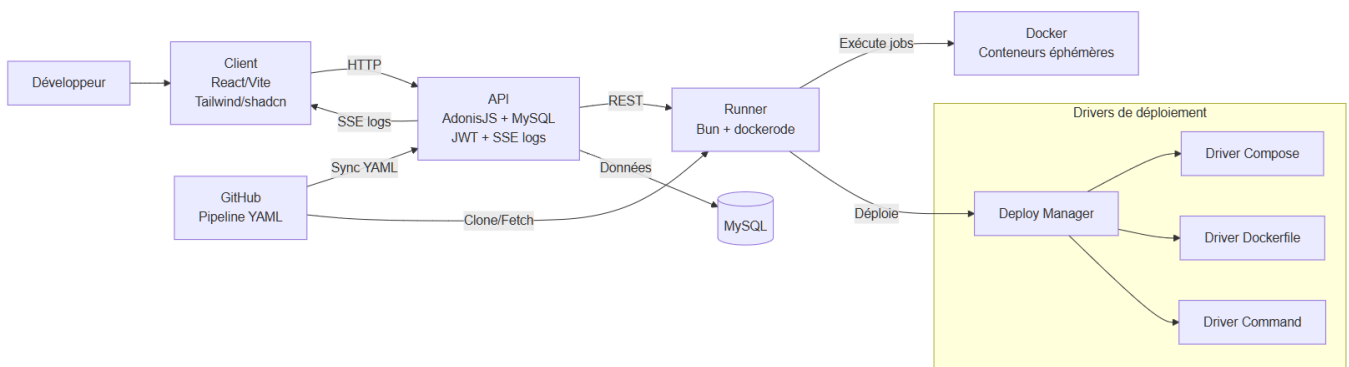
- le développement se fait dans **feature/\*** et est mergé dans **develop**,
- quand la release est prête, elle passe en **release/\***,
- puis on tag la version sur **main**, ce qui déclenche la partie **CD** (déploiement).

## 5. Architecture & Prototype

### 5.1 Vue d'ensemble

Shiply est un **mono-repo TypeScript** avec trois briques principales qui discutent entre elles.

### 5.2 Schéma architecture



### 5.3 Cycle d'un run (end to end)

1. **Sync pipeline** : côté UI, je demande la synchro YAML depuis GitHub → l'API récupère le fichier, le valide via **ajv** et le stocke.
2. **Lancement** : **POST /pipelines/:id/run** → l'API crée un **run** et ses **jobs** (status **pending**).
3. **Exécution** : le **runner** scrute l'API (heartbeat) et prend les jobs à faire.

Pour chaque job :

- clone/MAJ du **workspace** du projet,
- création d'un conteneur Docker (envs par défaut : **CI=1**, **TERM=dumb**, **NO\_COLOR=1**, **FORCE\_COLOR=0**),
- exécution de la commande, **stream des logs** vers l'API (SSE dispo côté lecture),
- maj du statut (**running** → **success** / **failed** / **canceled**).

4. **Agrégation** : l'API agrège l'état du run à partir de ses jobs.
  5. **Déploiement** (option) : un job spécial **\_\_shiply\_deploy\_\_:<driver>** déclenche le **Deploy Manager** (compose / dockerfile / command).
  6. **Post-checks** : smoke test simple.
- Consultation** : l'UI **pré-charge** les logs existants puis passe en **SSE** pour le live.

## 5.4 Prototype (écrans clés)

- **Projects** : liste, création, suppression (cascade DB + nettoyage workspace).
- **Project Details** : config du repo (fullName, branche par défaut, `pipelinePath`, `rootPath`), bouton **Sync YAML** (+ affichage validation).
- **Run Details** : timeline des jobs, badges de statut, panneau **Logs live** (SSE) avec fallback polling.
- **Deploy** : visibilité du job de déploiement, sortie des commandes, lien de vérif/URL publiée.

## 5.5 Modèle de données (résumé)

- **users** (id, email, passwordHash, tokens JWT)
- **projects** (id, name, repositoryFullName, defaultBranch, pipelinePath, rootPath, secrets chiffrés)
- **pipelines** (id, project\_id, yaml source, schemaVersion, validé ou non)
- **runs** (id, pipeline\_id, status, started\_at, finished\_at, triggered\_by)
- **jobs** (id, run\_id, name, image, command, status, started\_at, finished\_at, exit\_code)
- **logs** (id, job\_id, chunk\_seq, data) — pré-chargement possible via `GET /logs/:id`
- **deployments** (id, run\_id, driver, params, smoke\_result, url publiée)

## 5.6 Décisions techniques marquantes

- **SSE plutôt que WebSocket** pour les logs : plus simple côté infra (pas de broker), suffisant pour du flux de logs
- **dockerode** pour piloter Docker et garder un contrôle fin.
- **Types partagés** (`shared/`) pour garantir que l'UI et le runner lisent/écrivent les mêmes contrats (moins d'"off-by-type").
- **ajv** pour valider le YAML dès l'ingestion → on évite de découvrir des erreurs au moment de l'exécution.
- **Annulation coopérative** : si un run passe `canceled`, le runner arrête proprement les conteneurs liés.

## 5.7 Sécurité & perf côté archi

- **CORS strict** + en-têtes sécurisés (Nginx/Adonis).
- **JWT** pour l'auth et contrôle d'accès basique (RBAC étendu en perspective).
- **Masquage des secrets** dans les logs (filtrage serveur).
- **Workspaces séparés** par projet pour éviter les fuites de fichiers entre jobs.
- **Images Docker minimales** quand c'est possible pour réduire surface d'attaque et temps d'exécution.



## 6. Tests unitaires

---

### 6.1 Objectifs

Les tests unitaires jouent un rôle central dans la stabilité de Shiply. Leur but est de **détecter les régressions dès qu'une modification est introduite** et de sécuriser les parties critiques du code, notamment l'authentification, le chiffrement, l'exécution Docker et le streaming des logs en temps réel (SSE).

Dans la pipeline d'intégration continue, une règle est imposée : au moins **80 % des modules impactés** doivent être couverts par des tests, ce qui garantit que les évolutions n'affaiblissent pas la qualité globale.

---

### 6.2 Frontend — Vitest et Testing Library

Pour le frontend, j'utilise **Vitest** associé à **React Testing Library** et **user-event**. L'objectif est de tester les **composants UI** (par exemple l'affichage correct d'un statut ou d'un message d'erreur), les **slices Redux** qui gèrent l'état de l'application, ainsi que les **hooks et utilitaires**.

La stratégie combine :

- des **tests unitaires rapides**, centrés sur un composant ou une fonction,
  - quelques **tests d'intégration légers**, par exemple en montant une page complète avec son store pour vérifier que les interactions (clic, saisie, navigation) fonctionnent correctement.
- 

### 6.3 API — Japa + Supertest

Côté API, les tests sont écrits avec **Japa** comme test runner et **Supertest** pour envoyer des requêtes HTTP simulées. Les données de test sont générées via **Lucid ORM** et ses **factories**, ce qui permet de préparer rapidement des utilisateurs, projets et runs factices.

Pour la base de données :

- en local ou dans GitHub Actions, j'utilise un **MySQL dédié** via un service Docker,
  - pour des tests plus unitaires (par exemple sur les validators ou services isolés), je peux utiliser **SQLite en mémoire**, ce qui accélère énormément l'exécution.
-

## 6.4 Runner — Vitest avec mocks Dockerode

Pour le runner, qui s'appuie sur Docker, j'ai choisi de ne pas lancer de vrais conteneurs dans les tests unitaires.

À la place, j'utilise Vitest avec des mocks de dockerode (createContainer, start, logs, etc.).

Cette approche permet de vérifier que la configuration des conteneurs est correctement construite, que les variables d'environnement par défaut (par exemple `CI=1` et `TERM=dumb`) sont bien injectées automatiquement, que l'annulation coopérative fonctionne correctement lorsqu'un run est stoppé, et enfin que le mapping des volumes entre le workspace hôte et le conteneur est conforme aux attentes.

## 6.5 Exemple de tests

```
test.group('Runners claim & finish', () => {
  test('claim a queued job and finish it', async ({ client, assert }) => {
    const token = await getAuthToken(client)

    // Create project & pipeline with one step
    const proj = await client
      .post('/projects')
      .header('Authorization', `Bearer ${token}`)
      .json({ name: 'ClaimProj', key: `CLM_${Date.now()}` })
    proj.assertStatus(201)
    const projectId = proj.body().data.id as number

    const pipe = await client
      .post(`/projects/${projectId}/pipelines`)
      .header('Authorization', `Bearer ${token}`)
      .json({
        name: 'p',
        yaml: `version: 1\nname: p\nstages:\n  - name: build\n    steps:\n      - run: echo "hi"`,
      })
    pipe.assertStatus(201)
    const pipelineId = pipe.body().data.id as number

    const run = await client
      .post(`/pipelines/${pipelineId}/run`)
      .header('Authorization', `Bearer ${token}`)
      .json({})
    run.assertStatus(201)

    // Runner heartbeats and claims
    const hb = await client.post('/runners/heartbeat').json({ name: 'runner-A' })
    hb.assertStatus(200)

    const claim = await client.post('/runners/claim').json({ name: 'runner-A' })
    claim.assertStatus(200)
    assert.ok(claim.body().data)
    const jobId = claim.body().data.id as number

    // Finish the job
    const fin = await client.post(`/jobs/${jobId}/finish`).json({
      status: 'success',
      exitCode: 0,
      logsLocation: '/tmp/log',
      artifactsLocation: '/tmp/art',
    })
    fin.assertStatus(200)
    assert.equal(fin.body().data.status, 'success')
  })
})
```

## 6.6 Resultat de test

Voici les résultats d'une pipeline CI/CD github d'une serie de test sur l'api, via une base de donnée dédiée au test :

The screenshot displays a GitHub Actions workflow run interface. The workflow is titled "ci/release: run API tests with Bun and test env; remove GitHub OAuth..." and is currently in a "succeeded" state. The left sidebar shows the workflow steps: "Summary", "Jobs", "Run details", "Usage", and "Workflow file". The "Jobs" section is expanded, showing a list of jobs: "SonarQube Analysis", "API tests (Bun)", and "Build and push Docker images". The "API tests (Bun)" job is selected, and its details are shown in the main panel. The job is titled "API tests (Bun)" and "succeeded now in 55s". The main panel displays a list of steps for this job, including "Set up job", "Run actions/checkout@v4", "Set up Bun", "Install dependencies (workspace)", "Create test env file", and "Run DB migrations (fresh)". The "Run API tests" step is expanded, showing a list of commands and their output. The output includes a series of checks and logs, such as "Run bun run test", "Node ace test", "info [ booting application to run tests...", "api / Auth (tests/auth.spec.ts)", "register then login (2s)", "api / Logs SSE (tests/logs\_sse.spec.ts)", "strict mode: use allUnionTypes to allow union type keyword at '#/properties/version' (strictTypes)", "SSE stream returns appended logs (3s)", "api / Pipelines YML validation (tests/pipelines.yml\_validation.spec.ts)", "strict mode: use allUnionTypes to allow union type keyword at '#/properties/version' (strictTypes)", "rejects invalid yaml (no stages) (1s)", "api / Projects, Pipelines, Runs (tests/projects\_pipelines\_runs.spec.ts)", "strict mode: use allUnionTypes to allow union type keyword at '#/properties/version' (strictTypes)", "create projects, create pipeline, trigger run, get run (5s)", "api / Runners claim & finish (tests/runners\_claim\_finish.spec.ts)", "strict mode: use allUnionTypes to allow union type keyword at '#/properties/version' (strictTypes)", "claim a queued job and finish it (4s)", "api / Runners & Metrics (tests/runners\_metrics.spec.ts)", "runner heartbeat updates and metrics are exposed (478.82ms)", "api / Runners jobs listing and aggregation (tests/runners\_jobs\_listing.spec.ts)", "strict mode: use allUnionTypes to allow union type keyword at '#/properties/version' (strictTypes)", "list jobs of a run and get aggregated status (3s)", "PASS", "Tests 7 passed (7)", and "Time 28s".

## 7. Sécurité & Accessibilité

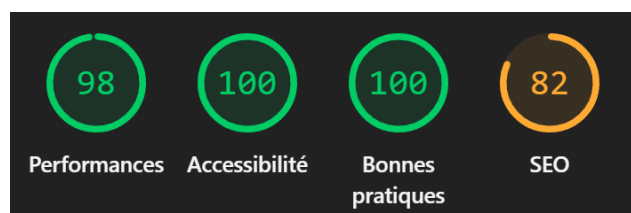
### 7.1 Objectifs

Deux axes principaux :

- **Sécurité** : protéger les secrets, limiter la surface d'attaque, contrôler les permissions Docker et éviter toute fuite dans les logs.
- **Accessibilité (A11y)** : garantir une interface utilisable au clavier, avec des contrastes corrects et une gestion claire des états (par exemple l'affichage des logs en temps réel).

Ces objectifs sont appliqués dès le MVP,

ce qui me permet d'obtenir un score d'accessibilité de 100.(voir annexe)



### 7.2 API (AdonisJS) — protections principales

Côté API, plusieurs mesures renforcent la sécurité :

- **Authentification** : l'accès se fait via des **JWT** (Bearer). Les tokens ont une durée de vie courte, avec vérification stricte.
- **CORS** : la configuration est volontairement stricte, seules les origines autorisées explicitement peuvent accéder, les credentials sont désactivés et les méthodes HTTP sont limitées.
- **En-têtes de sécurité** : un ensemble de headers est ajouté par Nginx et AdonisJS pour réduire les vecteurs d'attaque. Par exemple :
  - **Content-Security-Policy**: `default-src 'self'; img-src 'self' data;; connect-src 'self'` (bloque le chargement de ressources externes non prévues),
  - **X-Content-Type-Options**: `nosniff` (évite la détection abusive de type MIME),
  - **Referrer-Policy**: `no-referrer` (ne pas transmettre l'URL source),
  - **Permissions-Policy**: `geolocation=(), camera=(), microphone=()` (désactivation explicite des permissions sensibles),
  - **Cache-Control**: `no-store` pour les endpoints sensibles et le SSE.

- **Validation des données** : chaque payload est validé avant traitement. Le YAML de pipeline est contrôlé via **ajv** et les autres données via les validateurs d'Adonis. Cela empêche l'injection et limite les erreurs de format.
- **Rate limiting** : certaines routes sensibles (comme **POST /auth/login** ou **POST /pipelines/:id/run**) sont protégées par un middleware de throttling.
- **SSE sécurisé** : le flux temps réel des logs (**/jobs/:id/logs**) est protégé par un JWT transmis en **Authorization header**. Aucun token n'est mis en query string pour éviter la fuite dans les logs ou les proxys.

## 7.3 Runner & Docker — points de vigilance

Côté runner et Docker :

- **Pas de conteneur privilégié** : aucun job n'est lancé avec **--privileged** ou avec des capacités Linux (**--cap-add**) inutiles.
- **Utilisateur non-root** : quand l'image de base le permet, les jobs tournent sous un utilisateur non-root afin de réduire les risques d'escalade.
- **Filesystem restreint** : si possible, les conteneurs sont lancés avec **readOnlyRootFilesystem**. Les seuls montages autorisés sont ceux du **workspace** projet, ce qui évite l'accès à l'hôte.
- **Ressources limitées** : des quotas CPU et mémoire sont appliqués aux jobs, afin qu'un conteneur ne puisse pas saturer la machine.
- **Réseau isolé** : les jobs sont exécutés sur un réseau Docker dédié. Les conteneurs n'ont pas accès au socket Docker de l'hôte, ce qui empêcherait d'exécuter des commandes non prévues.
- **DOCKER\_HOST sécurisé** : par défaut, le runner utilise le socket Unix local. Si jamais un daemon TCP est exposé (ex. Windows avec Docker Desktop), il doit être limité à **localhost** ou sécurisé par TLS.

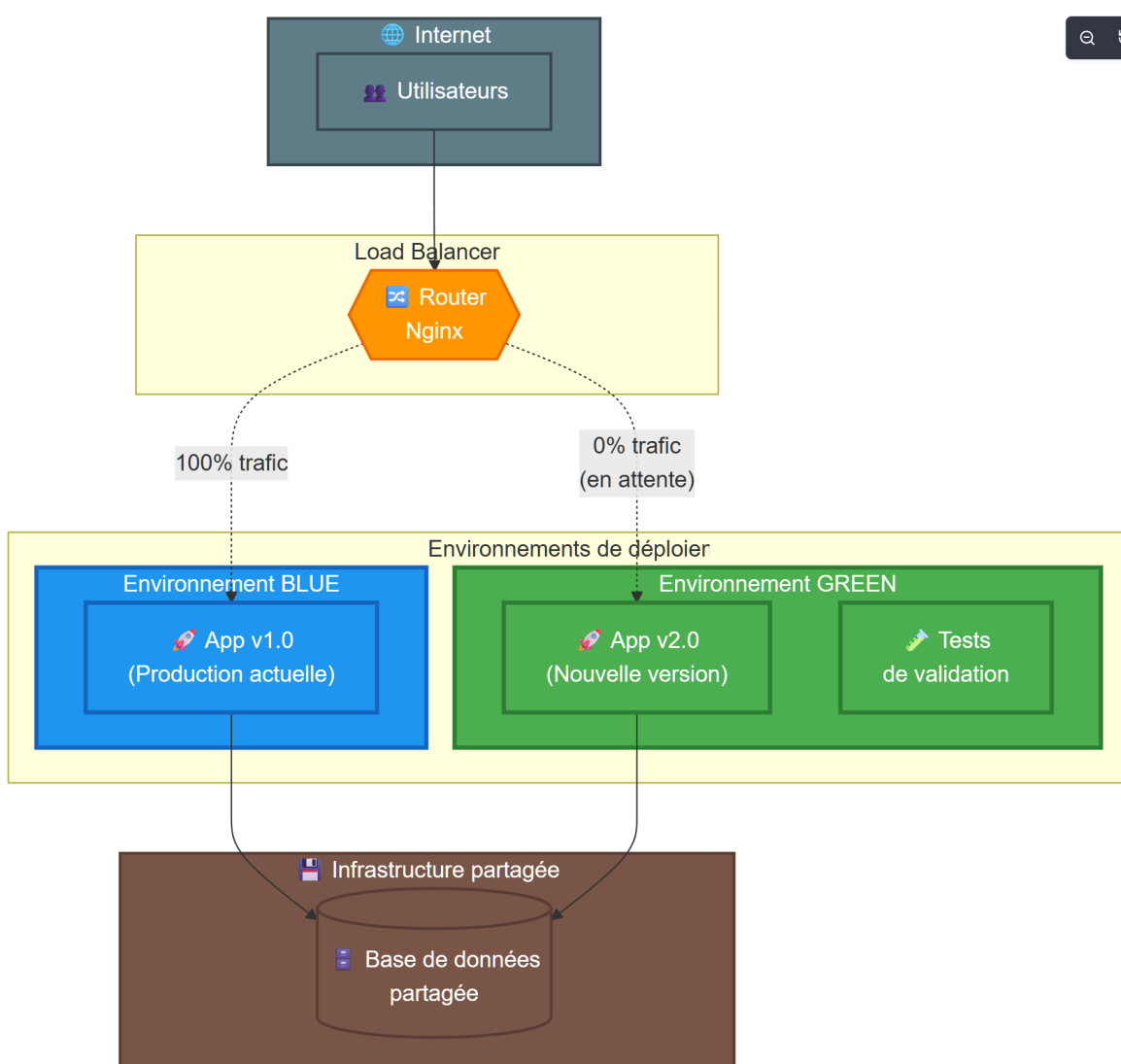
## 8. Déploiement progressif

### 8 Stratégies retenues (MVP)

Je pars sur une approche simple avec **Docker + Nginx**, sans passer par un orchestrateur lourd comme Kubernetes.

L'idée est d'utiliser une stratégie de déploiement **Blue-Green** : **deux versions** de l'application tournent en **parallèle**, la version *blue* (**stable**) et la version *green* (**nouvelle**). Quand la nouvelle version est prête, je fais un smoke test pour vérifier que tout fonctionne correctement.

Si le test est concluant, je bascule le trafic de Nginx de *blue* vers *green*. En cas de problème, il suffit de revenir rapidement à la version stable en repassant sur *blue* via un simple switch et un reload de Nginx.



## 9. Cahier de recettes

---

Ce cahier décrit **comment valider le MVP de Shiply de bout en bout**. L'idée est d'expliquer des parcours réalistes, ce qu'on doit voir, et comment conclure si c'est valide ou non.

### 9.1 Préparation

La plateforme tourne (prod ou dev), l'API répond sur `/health`, et un compte utilisateur pour se connecter. Le runner est démarré et communique (heartbeat) auprès du serveur.

### 9.2 Recette A — Authentification (login/logout)

On ouvre l'UI et on se connecte avec un compte existant. Si tout va bien, on est redirigé vers la page **Projects** et le token JWT est bien utilisé sur les appels API. Verification que `/health` renvoie `200` depuis l'onglet réseau, puis je me déconnecte : on revient à **Login** et les appels protégés deviennent `401`. Conclusion : l'auth est ok si le cycle login → appels autorisés → logout → `401` se déroule comme prévu.

### 9.3 Recette B — Création d'un projet + synchronisation YAML

Je crée un **Project** depuis l'UI avec : nom lisible, `repositoryFullName`, branche par défaut, `pipelinePath` et `rootPath`. Une fois créé, je clique **Sync YAML**. L'API va chercher le fichier depuis github, le **valide** (ajv) et enregistre la version courante. Je m'attends à voir « Pipeline valide » (ou un message d'erreur utile si le YAML est invalide).

**Critère de réussite** : après une synchro valide, la page **Project Details** affiche la version du pipeline et propose le bouton **Run**.

### 9.4 Recette C — Lancer un run et suivre les logs

Depuis le projet, je lance **Run pipeline**. L'API crée un **run** avec des **jobs** en `pending`. Le runner récupère le premier job, crée un conteneur `alpine:3.19`, exécute la commande et **stream** les logs. L'UI pré-charge ce qui existe puis passe en **SSE** pour le direct ; je dois voir « BUILD OK », puis « TEST OK ». À la fin, les deux jobs sont `success`, et le **run** agrégé aussi.

**À surveiller** : dans l'onglet Réseau, le flux `/jobs/:id/logs` a bien `Content-Type: text/event-stream`. Si je coupe le SSE (simulateur réseau), l'UI doit continuer en **polling** sans planter : les lignes de logs s'affichent quand même, juste moins "en live".

### 9.5 Recette D — Job de déploiement (simulation "command")

Dans le même run, le dernier job est `__shiply_deploy__:command`. Ici, je n'envoie pas en prod, je **simule**. Le job doit exécuter le script qui affiche « Déploiement ok » La sortie doit apparaître dans les logs, et je peux vérifier côté serveur que le fichier existe bien dans le répertoire de workspace du projet.

**Conclusion** : le Deploy Manager est câblé (paramètres reçus, logs renvoyés, statut `success`).

## 10. Plan de correction des bogues (Redmine)

---

**Principe.** À chaque bug : je l'enregistre, je le reproduis, je corrige avec un test, je passe la CI, je release, je déploie. S'il y a un souci, je rollback. Tout est tracé dans **Redmine**.

### Étapes

1. **Ticket Redmine.** Tracker *Bug*, sévérité (S0→S3), composant (client/api/runner), version affectée, lien vers le run/job et les logs.
2. **Reproduction.** Cas minimal reproductible (YAML simple ou scénario UI). Objectif : avoir un résultat stable (exit code, message).
3. **Correctif + test.** J'écris d'abord un test qui échoue (Japa/Vitest), je corrige, le test passe.
4. **Revue + CI.** PR liée au ticket ([refs #123](#)). La CI doit tout valider : typecheck, lint, tests, build, Trivy, LHCI.
5. **Release & déploiement.**
  - Urgent (S0/S1) : branche **hotfix/** depuis [main](#), tag **patch**, CD déploie.
  - Sinon : merge sur [develop](#), inclus dans la prochaine release.
6. **Rollback (si besoin).** On redéploie le **tag précédent** et on note l'heure/impact dans le ticket.
7. **Clôture.** Le ticket passe "Livré → Clos" avec la version corrigée. En 5 lignes max, je note la **cause** et l'**action préventive** (ex : test ajouté, règle de validation durcie).

### Règles rapides

- Un bug = **un test** ajouté (régression évitée).
- Pas de secret dans les logs du ticket.
- Message de commit clair : [fix: ... \(fixes #123\)](#).

### Critères d'acceptation (pour validé)

- Le **scénario utilisateur** qui posait problème fonctionne.
- La **CI est verte** et le **/health** est OK après déploiement.



# 11. Manuel de déploiement

---

Cette partie explique comment déployer **Shiply** en **développement** et en **production**.

## 11.1 Principes généraux

Shiply est composé de trois briques : une **API AdonisJS**, un **front React** servi par **Nginx**, et un **runner** qui exécute les jobs **Docker**.

En production, les images sont tirées depuis **Docker Hub** et orchestrées via `docker-compose.prod.yml`.

La configuration s'appuie sur deux fichiers d'environnement : un `.env` à la **racine** (chemin des workspaces) et un `server/.env` pour l'API.

### Pré-requis

- **Docker** et **Docker Compose** installés et accessibles.
- Accès **MySQL** (hôte/port/utilisateur/mot de passe/base) et ports ouverts.
- Sous **Windows**, Docker Desktop peut nécessiter l'exposition du daemon TCP (`DOCKER_HOST`, voir Dépannage).
- (Perso) Je peux utiliser **spleendb.fr** (mon app) pour créer et gérer les bases MySQL.

### Configuration des environnements

#### Racine (`.env`)

```
# Windows (Docker Desktop)
HOST_WORKSPACE_DIR=/run/desktop/mnt/host/d/Dev/Perso/deploy

# Linux
HOST_WORKSPACE_DIR=/app/deploy
```

API (server/.env)

```

NODE_ENV=production
HOST=0.0.0.0
PORT=3333

# Clé d'application (obligatoire) – générer avec: node ace key:generate
APP_KEY=base64:...ou:hex...
APP_NAME=Shiply
LOG_LEVEL=info

# Base de données MySQL
DB_HOST=your_mysql_host
DB_PORT=3306
DB_USER=your_user
DB_PASSWORD=your_password
DB_DATABASE=shiply

# (Optionnel) OAuth GitHub
GITHUB_CLIENT_ID=
GITHUB_CLIENT_SECRET=
GITHUB_CALLBACK_URL=
```

## 11.2 Déploiement local (dev)

Installer les dépendances, lancer le monorepo :

- bun install
- bun run dev

Appliquer les migrations (dans le conteneur API) ainsi que les fausses données

- cd server
- node ace migration:fresh --seed

Vérifier la santé

```
curl -f http://localhost:12001/health || exit 1
```

## 12. Manuel de mise à jour

---

Cette partie décrit comment préparer et publier une nouvelle version de Shiply.

### Principes généraux

- Gestion de versions avec **GitFlow** : les branches `release/*` partent de `develop`, et les versions suivent le **semver** (x.y.z).
- Un **changelog** est généré automatiquement à partir des **Conventional Commits**.
- Les **images Docker** (API et front) sont publiées sur Docker Hub, et utilisées en production via `docker-compose.prod.yml`.
- La **CI** vérifie : typecheck, lint, tests ( $\geq 80\%$  des modules touchés), et scans qualité/sécurité (Trivy, LHCI).

### Prérequis

- `.env` à la racine avec `HOST_WORKSPACE_DIR` (voir manuel de déploiement pour Windows/Linux).
- `server/.env` avec au minimum `APP_KEY` et les variables `DB_*`.
- Accès Docker et au serveur cible.

### Étapes d'une release

#### 1) Préparation de la branche et du changelog

```
git checkout develop && git pull --ff-only
git checkout -b release/x.y.z
npx --yes conventional-changelog-cli -p angular -i CHANGELOG.md -s
git add CHANGELOG.md && git commit -m "docs(changelog): update for x.y.z"
git push -u origin release/x.y.z
```

#### 2) Publication de la version (après merge sur `main`)

```
git checkout main && git pull --ff-only
git tag -a vx.y.z -m "release vx.y.z"
git push origin vx.y.z
```

La CI construit et pousse automatiquement les images Docker :

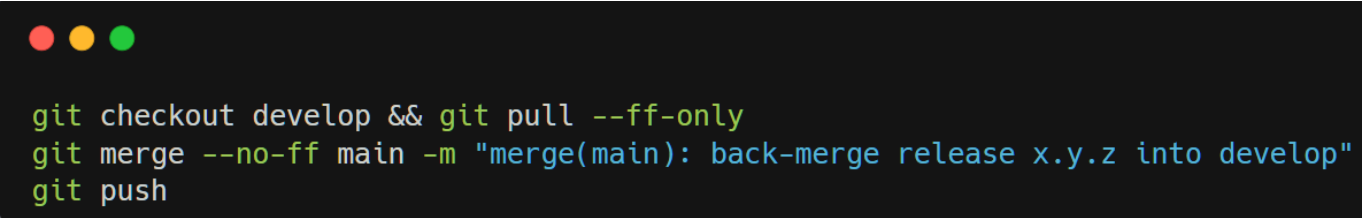
- API : `doxteurn/shiply-api:vx.y.z`
- Web : `doxteurn/shiply-nginx:vx.y.z`

### 3) Mise à jour en production

La CD se déclenche automatiquement après le push du tag :

- Pull des nouvelles images,
- Redémarrage des services via `docker-compose`,
- Vérification de l'état de santé,
- Front accessible sur <https://shiply-app.fr/> une fois le déploiement termin

### 4) Ré-alignement des branches



```
git checkout develop && git pull --ff-only
git merge --no-ff main -m "merge(main): back-merge release x.y.z into develop"
git push
```

## 13. Manuel d'utilisation

---

Shiply est une plateforme CI/CD accessible depuis un navigateur. Elle permet de créer des projets, de synchroniser un pipeline YAML depuis GitHub et de lancer des exécutions dans des conteneurs Docker.

### 13.1 Accès à l'application déployée

Shiply est actuellement déployé et accessible en ligne à l'adresse suivante :

👉 <https://shiply-app.fr/>

Pour les besoins du projet, un compte administrateur est mis à disposition :

- **Email** : `admin@admin.com`
- **Mot de passe** : `password`

Ce compte permet de se connecter à l'interface, de créer des projets, de synchroniser un pipeline et de lancer des exécutions.

Il est destiné uniquement aux tests et démonstrations dans le cadre du projet, le mot de passe simplifié pour cette raison.

### 13.2 Créer et configurer un projet

Depuis la page « Dashboard », créer un nouveau projet puis sélectionner GitHub comme source. Renseigner :

- **Branche par défaut** (`defaultBranch`, ex: `main`),
- **Racine du projet** (`rootPath`) si le dépôt contient un sous-dossier,
- **Mode de déploiement** (`runMode`: `compose`, `dockerfile` ou `command`),
- **Variables d'environnement** (paires `key/value` utilisées par les jobs).

Une fois le projet enregistré, Shiply tente automatiquement de synchroniser le pipeline YAML depuis le dépôt.

---

### 13.3 Synchroniser une pipeline

La synchronisation lit le fichier YAML de la pipeline sur la branche par défaut (`.shiply.yml/.yaml` à la racine ou dans `rootPath`).

- Si erreur « fichier introuvable » : vérifier `repositoryFullName`, `defaultBranch`, `pipelinePath`, `rootPath`.
  - Une fois validé, le pipeline apparaît dans la fiche projet.
- 

### 13.4 Exécuter un pipeline (Run)

Depuis la fiche projet :

- Cliquer sur **Run pipeline**.
  - Chaque stage/step est exécuté par le runner dans des conteneurs Docker.
  - Le suivi est en temps réel : cliquer sur un job pour voir sa console.
  - Le bouton **Stop** annule proprement le run.
  - Le statut global (`running`, `success`, `failed`, `canceled`) s'agrège automatiquement.
- 

### 13.5 Logs en temps réel

- Les logs sont accessibles depuis la page du run.
  - **Préchargement** : les logs existants sont d'abord affichés.
  - **Streaming** : le flux SSE affiche les nouvelles lignes en direct (fallback en polling si besoin).
  - **Nettoyage** : un bouton « Clear » vide seulement l'affichage côté navigateur.
-

## 13.6 Déployer un run réussi

Si un run est **success**, le bouton **Deploy** est disponible.

Il déclenche un job spécial `__shiply_deploy__:<driver>` qui :

- utilise le driver défini (`compose`, `dockerfile`, `command`),
  - stream les logs comme un job classique,
  - peut exécuter un **smoke test** basique.
- 

## 13.7 Supprimer un projet

Depuis la fiche projet, cliquer sur **Delete project**. Cela supprime :

- les pipelines, runs, jobs associés,
  - le workspace correspondant côté serveur.
- 

## 13.8 Astuces & dépannage

- **Sync GitHub 404** : vérifier `repositoryFullName`, `defaultBranch`, `pipelinePath`, `rootPath`.
- **Runner/Docker (Windows)** : si les conteneurs ne démarrent pas, définir `DOCKER_HOST=tcp://localhost:2375` dans Docker Desktop.
- **Statuts figés** : l'UI se rafraîchit automatiquement mais un reload peut aider si la connexion réseau a coupé.

## 14. Choix techniques

---

Dès le début du projet, j'ai fait le choix de travailler avec des technologies que je maîtrise déjà, afin d'assurer à la fois **rapidité de développement** et **maintenabilité**.

### 14.1 Perspectives d'évolution

- **Frontend : React + Vite + TypeScript + Tailwind + shadcn/ui**

J'ai choisi React car c'est une librairie front très répandue, avec un écosystème populaire. Vite apporte une expérience de dev rapide (HMR quasi instantané). TypeScript sécurise le code avec un typage strict, ce qui est important pour un projet un peu ambitieux comme Shiplly. Enfin, Tailwind et shadcn/ui m'ont permis de construire une interface propre, réactive et accessible rapidement, sans réinventer la roue côté design système.

- **Backend : AdonisJS (Node.js/TypeScript)**

J'ai retenu AdonisJS car c'est un framework structuré (similaire à Laravel côté PHP), qui propose une organisation claire (controllers, services, validators) et intègre nativement un ORM (Lucid) pour MySQL. Cela m'évite de devoir assembler plein de librairies disparates et me donne un cadre solide. L'utilisation de JWT pour l'authentification est cohérente avec une architecture SPA.

- **Runner : Bun + dockerode**

Le runner est une pièce critique : il exécute les jobs dans des conteneurs éphémères. J'ai utilisé **Bun** car il démarre très vite, consomme peu de ressources, et reste compatible avec l'écosystème npm. Pour dialoguer avec Docker, j'ai choisi **dockerode**, qui est simple et fiable. Le runner lance les steps dans des containers avec un montage du workspace et gère l'annulation proprement.

- **Base de données : MySQL**

Je me suis tourné vers MySQL pour sa stabilité et parce que je l'utilise déjà dans d'autres projets. C'est suffisant pour gérer les entités (users, projects, pipelines, runs, jobs, logs). L'intégration avec Lucid ORM facilite beaucoup la gestion des migrations.

- **Infrastructure : Docker & Docker Compose**

L'application est packagée dans des conteneurs et déployée via `docker-compose.prod.yml`. Ce choix rend le projet auto-hébergeable facilement : un simple `docker compose up -d` permet de lancer l'API, le frontend (servi par Nginx) et le runner. Les images sont publiées sur Docker Hub avec des tags versionnés (SemVer), ce qui garantit une traçabilité et simplifie les mises à jour et rollback.

- **Gestion de code : GitFlow + Conventional Commits**

J'ai suivi un workflow Git clair : `develop` pour l'évolution, `main` pour les versions stables, et des branches `feature/*`, `release/*`, `hotfix/*`. Les commits respectent les conventions (feat, fix, docs...), ce qui permet de générer automatiquement un changelog et d'avoir un historique lisible.



## 14.2 Perspectives d'évolution

Le MVP de Shiply fonctionne, mais plusieurs axes d'amélioration sont possibles pour aller vers un produit plus complet :

1. **Gestion avancée des droits (RBAC)**

Aujourd'hui, il y a seulement une authentification basique par compte utilisateur. Il faudrait ajouter des rôles (admin, maintenir, viewer) avec des permissions fines par projet.

2. **Artefacts et stockage**

Les jobs peuvent générer des fichiers (build, tests, rapports). Pour l'instant, je n'ai pas encore de système d'artefacts. L'idée serait d'ajouter un service d'upload/stockage (S3 ou équivalent) pour conserver et télécharger les résultats des runs.

3. **Notifications**

Intégrer des notifications (Slack, Discord, email) pour prévenir des résultats de runs ou des déploiements.

4. **Observabilité renforcée**

Bien que Shiply soit branché sur Grafana/Prometheus pour la base (health, métriques simples), il faudrait exposer plus de données (temps moyen de run, ressources consommées par conteneur, taux de succès/échec) et créer des dashboards complets.

## 15. Conclusion

---

Le projet **Shiply** m'a permis de concevoir et de développer une plateforme **CI/CD auto-hébergeable**, pensée pour répondre à mes besoins, mais aussi de répondre à un cadre académique.

L'objectif était de créer une solution capable de **définir, exécuter et suivre des pipelines** de manière simple et reproductible, en s'appuyant sur Docker pour l'isolation et sur une interface web pour la lisibilité.

Le résultat est un **MVP fonctionnel** :

- une API AdonisJS qui gère l'authentification, les projets, les pipelines et les runs,
- un runner Bun + dockerode qui exécute les jobs dans des conteneurs éphémères,
- un frontend React moderne qui permet de suivre en direct l'avancement et les logs.

Tout est orchestré dans un monorepo TypeScript, versionné via GitFlow, avec une CI/CD qui automatise la qualité (tests, lint, sécurité) et le déploiement en production. Le projet est aujourd'hui déployé en ligne (<https://shiply-app.fr>) et utilisable via un compte de démonstration.

Au-delà de la technique, ce projet m'a appris à gérer un cycle complet de développement : cadrage, architecture, intégration continue, déploiement continu, sécurité, accessibilité, documentation et maintenance. J'ai pu mettre en pratique des notions comme le **typage strict**, la **traçabilité par SemVer**, ou encore la gestion d'environnements via Docker et Compose.

Bien sûr, Shiply reste un MVP : il existe encore de nombreuses perspectives d'évolution, comme l'ajout d'un **système d'artefacts**, une **gestion avancée des rôles (RBAC)**, des **notifications externes** ou une **observabilité renforcée**. Mais la base est solide, et les choix techniques me permettront de faire évoluer le projet sans repartir de zéro.

En conclusion, Shiply atteint son objectif initial : proposer une solution CI/CD simple, auto-hébergeable et extensible, qui reflète les bonnes pratiques modernes de l'ingénierie logicielle et me donne une vraie expérience concrète de bout en bout.

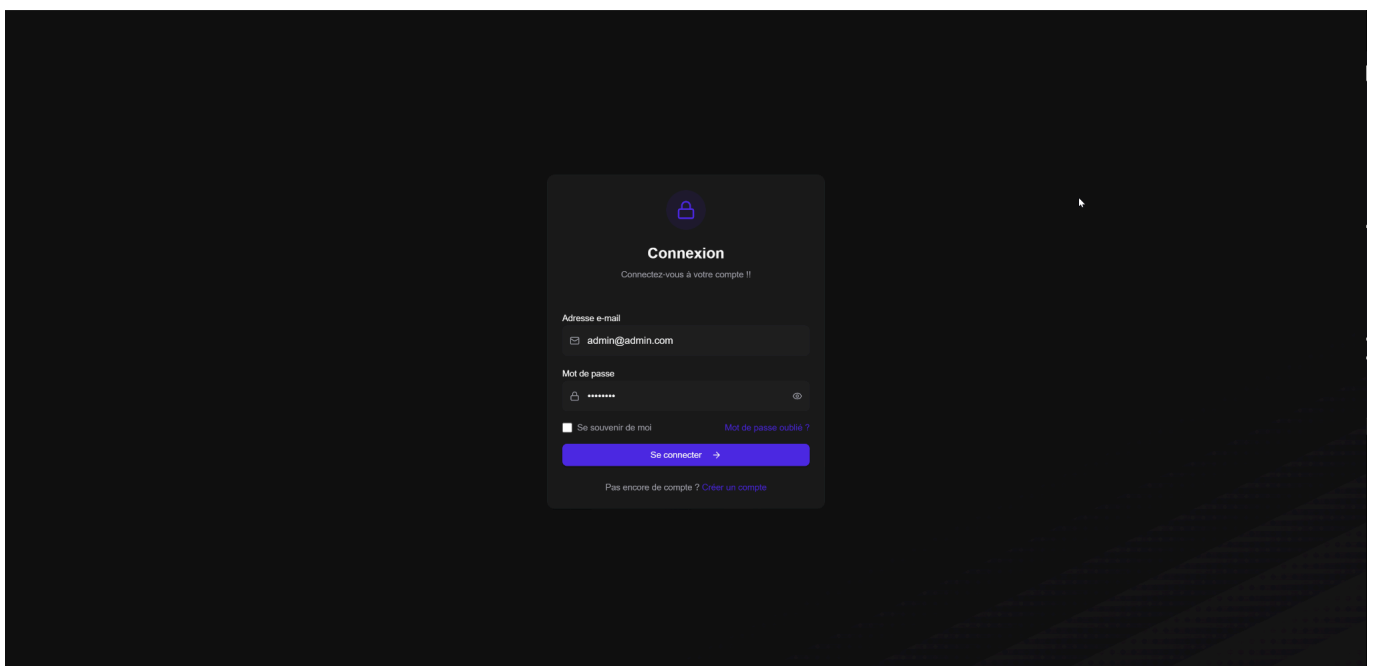
## 16. Annexes

---

Cette annexe rassemble quelques captures de l'interface Shiplly afin de donner un aperçu concret du fonctionnement de la plateforme. Elles ne remplacent pas le manuel d'utilisation, mais illustrent les pages principales.

### 1) Page de connexion

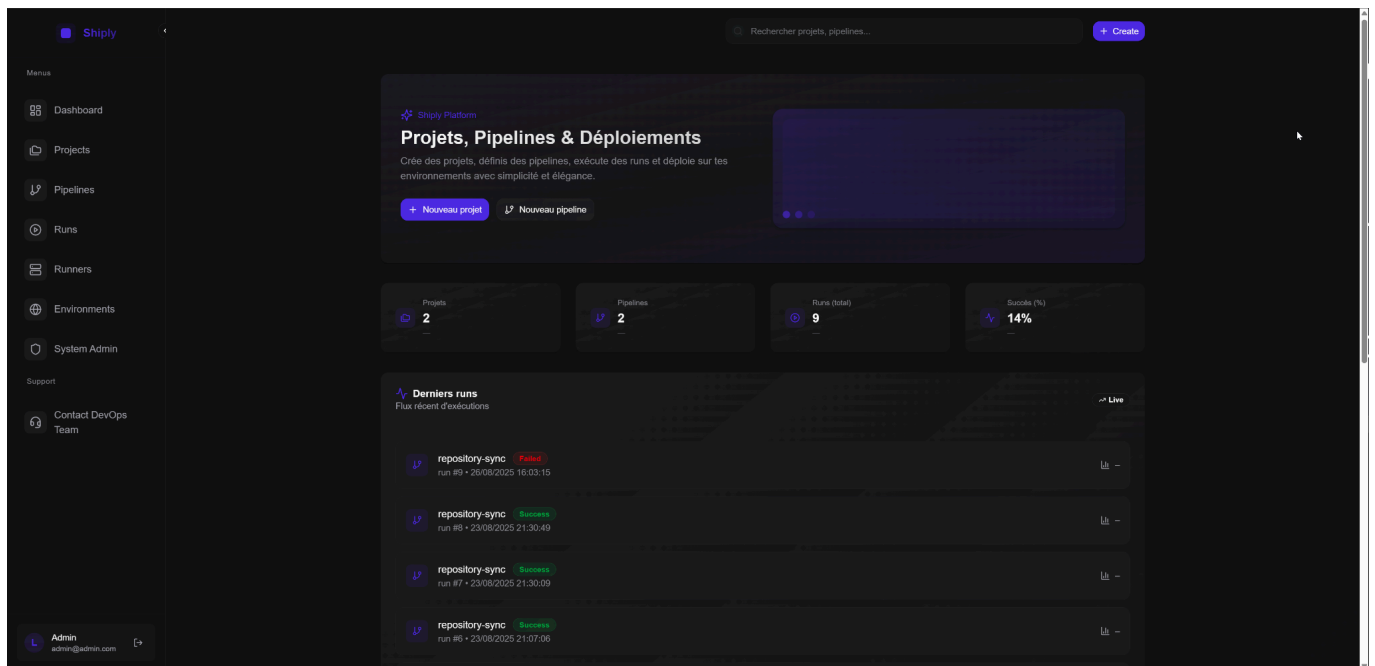
Capture de l'écran **Login**, permettant de se connecter avec un compte utilisateur ou le compte de démonstration `admin@admin.com`.



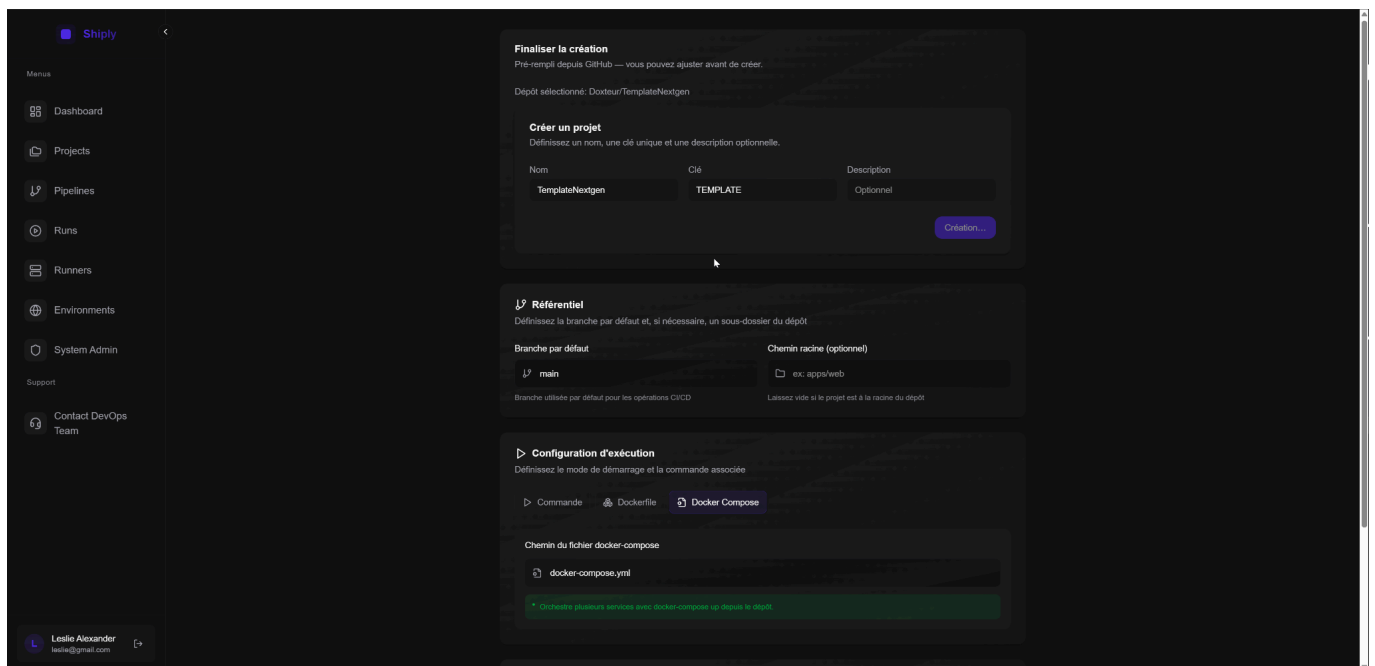
### 2) Tableau de bord (Dashboard)

Aperçu de la page d'accueil après connexion, qui liste les projets existants.

On y retrouve les informations essentielles : nom du projet, dépôt Git associé, branche par défaut et statut de la dernière exécution.

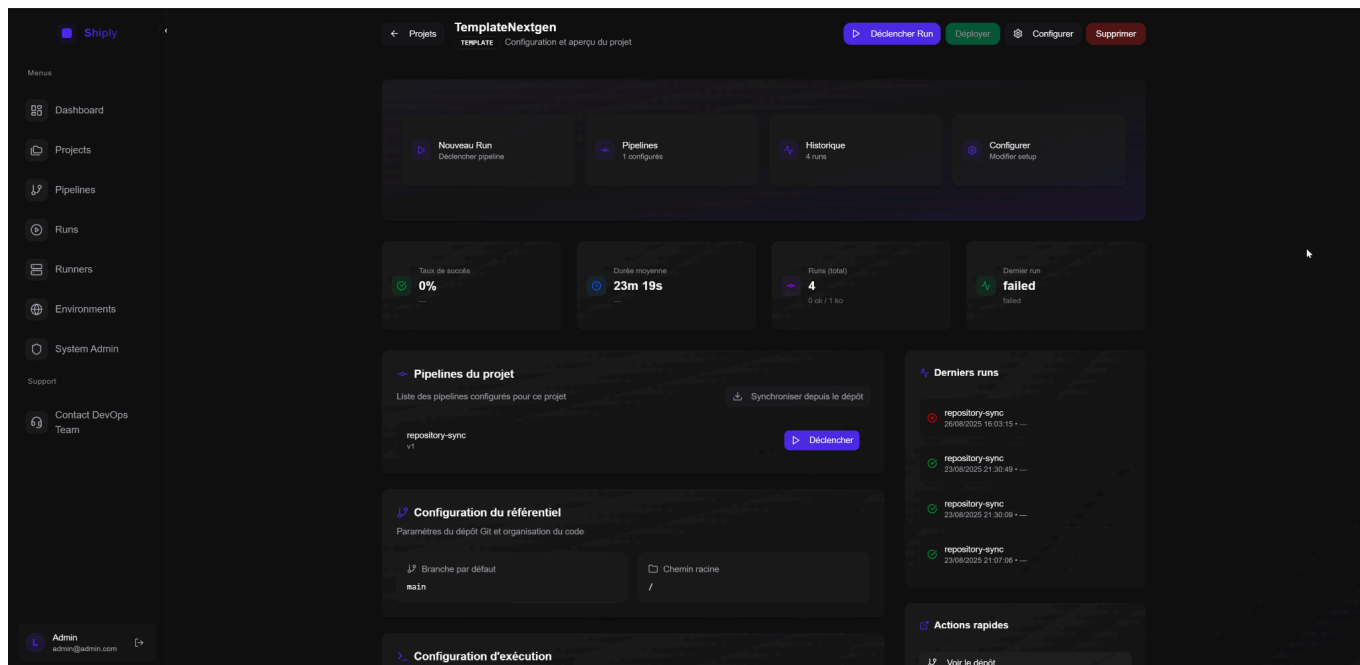


### 3) Création d'un projet



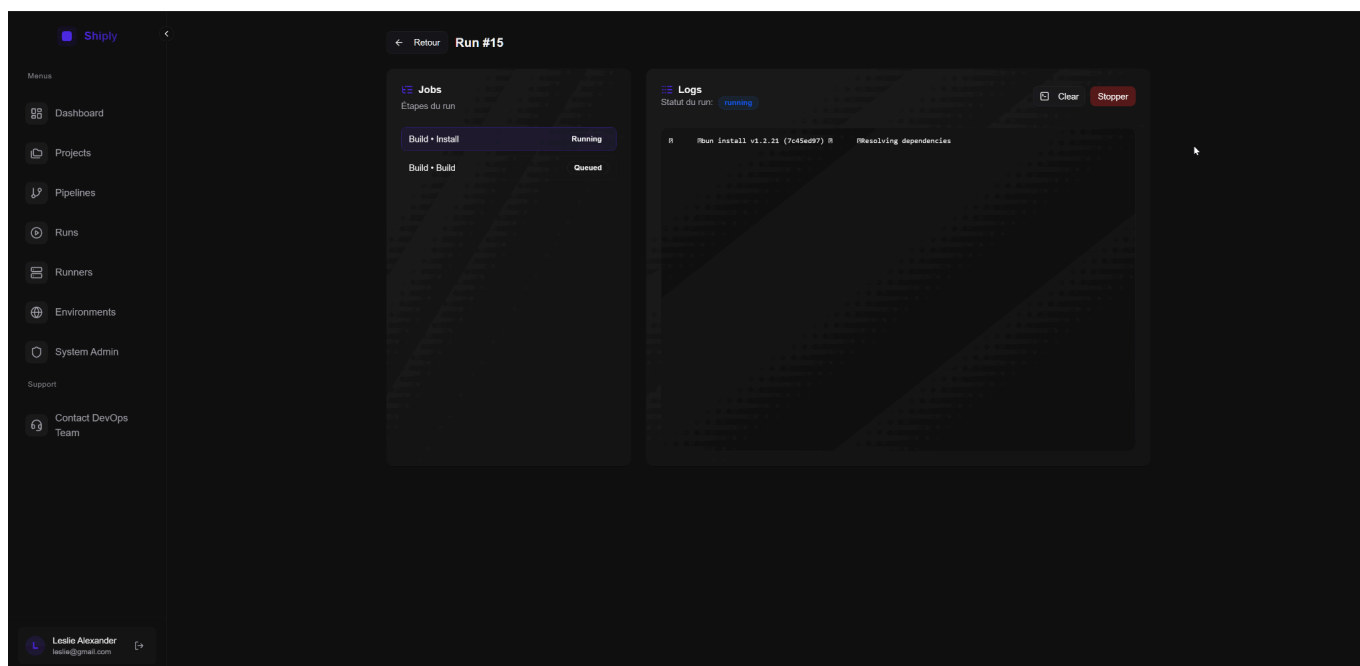
## 4) Détails d'un projet

La fiche projet montre la configuration (repository, pipeline path, runMode, variables d'environnement) et permet de synchroniser le pipeline ou de lancer une exécution. *(screenshot project details ici)*



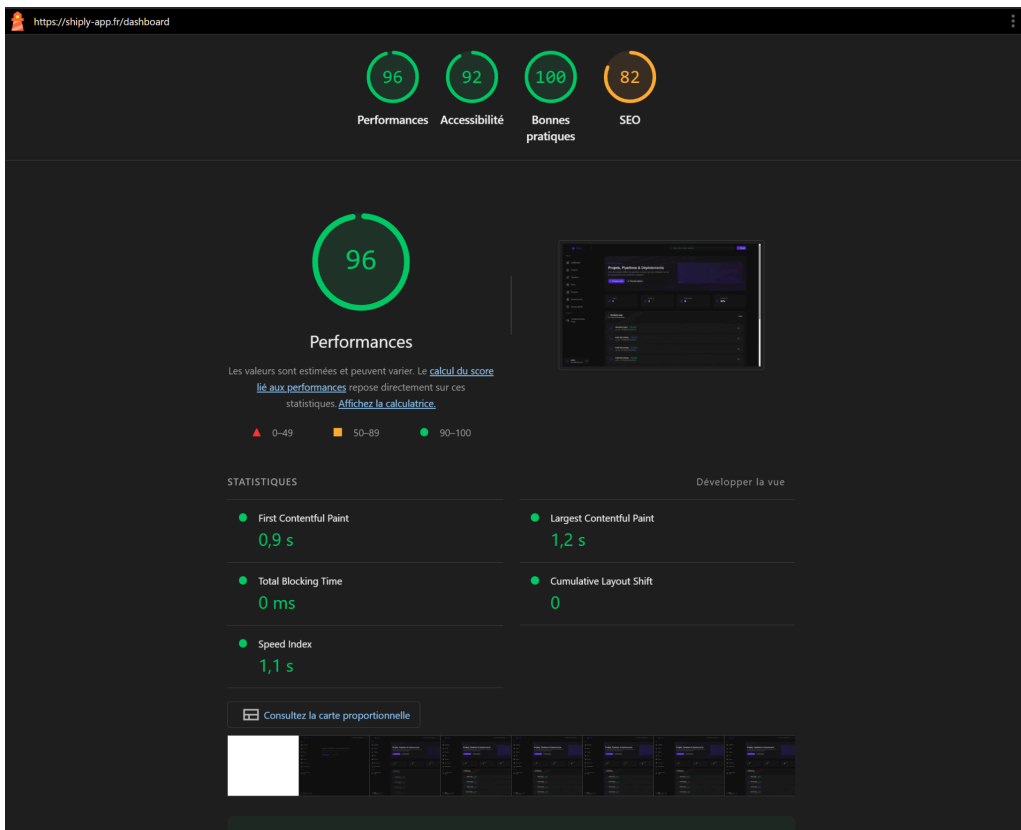
## 5) Suivi d'un run

Exemple d'un run : chaque job est affiché avec son statut, et les logs sont visibles en temps réel grâce au streaming SSE. *(screenshot run details ici)*



## 6) Rapport Lighthouse

### Rapport Dashboard



### Rapport Landingpage:

