



5IDEV

Projet d'intégration de développement

Thibault Six

10 chapitres

SUPPORTS DE COURS

<https://cours.tsix.be/cours/projet-dintegration-de-developpement>

Version 10 · 19/10/2026

PRÉSENTATION

5IDEV — Projet d'intégration de développement

Slides de présentation du cours

- [deck1-presentation-cours.pdf](#)

Pourquoi ce cours ?

Vous savez coder. Ce cours vous apprend à **livrer** : mettre une application en production, vite et proprement, en équipe, sur plusieurs stacks, avec les bonnes pratiques d'industrialisation (CI/CD, hébergement, documentation).

C'est aussi votre **répétition générale pour le travail de fin d'études** : cahier des charges, dossier technique, défense orale — tout ce qu'on vous demandera au TFE, vous allez le pratiquer ici trois fois.

Acquis d'apprentissage visés

Au départ d'une proposition de projet formulée par le chargé de cours, en exploitant un **framework backend** et un **framework frontend**, en intégrant le **responsive design** et en respectant les normes et standards en vigueur, vous serez capables :

1. de **produire et défendre un cahier des charges et son dossier technique** par rapport à la proposition du chargé de cours ;
2. d'**implémenter une base de données et l'intégrité des données** ;
3. de **déployer et justifier le site** répondant aux consignes figurant dans le cahier des charges.

Ce que nous allons faire

- **3 projets sur l'année** : P1 (25 %) · P2 (25 %) · P3 (50 %)
- **2 à 3 stacks différentes** au total — on ne refait pas deux fois la même chose
- Chaque projet part d'une **proposition du chargé de cours**, avec ses **exigences obligatoires** propres — elles seront annoncées au lancement de chaque projet
- **Travail en groupe** : groupes de 2 fortement recommandés, groupes de 3 autorisés à la discrétion du professeur. Solo interdit.
- **Chaque semaine** : un incrément déployé en production

- Après chaque projet : **2 semaines de congés** pour finaliser le travail et préparer la **présentation orale**, qui a lieu au premier cours après les congés

Rythme de travail

Le cours alterne deux types de séances :

Une semaine sur deux	L'autre semaine
Revue d'avancement : chaque groupe remet un rapport d'avancement (fonctionnalités implémentées + captures d'écran) dans le devoir Teams avant la séance , puis le présente en live. Feedback et conseils du professeur.	Point de théorie sur un sujet d'industrialisation : CI/CD, cache, analyse statique de code, serverless...

Méthode de travail

- **GitHub** : branches `feature/*` → Pull Request → review → `main` = production . Repo **public de préférence** (sinon, il faudra inviter l'enseignant à chaque projet).
- **CI/CD** : lint + build (tests si présents) à chaque PR, déploiement automatique sur `main` . On commence par la base, puis on intègre petit à petit de nouveaux outils pour améliorer la qualité du code au fil de l'année.
- **IA** : les assistants et agents IA sont autorisés. Règle d'or : **comprendre et valider chaque suggestion** (par des tests, notamment). Un carnet métacognitif sur votre usage de l'IA accompagne chaque projet.
- Un chapitre **GitHub Flow** est fourni pour apprendre à collaborer avec Git sans se marcher sur les pieds.

Livrables (par projet)

1. **URL de production publique** et fonctionnelle
2. **Repo GitHub** (accès enseignant)
3. **Rapport de projet au format PDF** (structure fournie — template disponible)
4. **Carnet métacognitif** sur l'utilisation de l'IA (template fourni)
5. **Présentation orale** de votre application devant la classe, par groupe

Règles non négociables

- Les **noms de tous les membres du groupe** figurent sur le rapport. Rapport sans noms = **0**.
- Le rapport est remis **au format PDF**. Autre format = **0**.

Je serai intraitable sur ces deux points.

Hébergement

Chaque projet doit être accessible sur une URL publique. Le choix de la solution d'hébergement fait partie du travail : vous la choisirez et la justifierez (une activité de recherche est prévue au premier cours). Les **secrets** passent par des variables d'environnement — jamais de `.env` commité.

Évaluation

- **Seuil de réussite** : les 3 acquis d'apprentissage doivent être acquis (grille fournie).
- **Degré de maîtrise** : architecture, interface, fonctionnalités, rapport technique, gestion de projet et **qualité de la présentation orale**.

-  [Fiche UE 2026-2027 \(PDF\)](#) 
-  [Grille d'évaluation \(PDF\)](#) 

Calendrier

Période	Projet
Septembre → vacances d'automne (~7 semaines)	Projet 1 (25 %)
Automne → Noël	Projet 2 (25 %)
Janvier → mi-mars	Projet 3 (50 %)

La remise de chaque projet tombe pendant les congés qui le suivent : vous avez ces **2 semaines** pour finaliser et préparer votre présentation orale, donnée au cours de la rentrée.

Ce cours et ses documents ont été écrits en collaboration avec Claude Fable 5 et GPT-5.6 Sol.

Sommaire

- 5IDEV — Projet d'intégration de développement

- 1 **Projet 1**

- 1.1 Hébergement

- 1.2 Workflow GIT

- 1.3 CI/CD

- 1.4 Lint & analyse statique

- 1.5 Où faire tourner votre application ?

- 1.6 Backups

- 2 **Projet 2**

- 2.1 Exemples d'applications & inspirations

- 2.2 Recherche Textuelle

CHAPITRE 1

Projet 1 : Messagerie temps réel (B2B)

Pondération : 25 % de la note annuelle · Remise : **02/11/2026 à 23h59** (lien GitHub + rapport PDF dans le devoir Teams) · Présentation orale au premier cours après les congés d'automne

1. Objectif & périmètre

Vous allez construire une **application de chat temps réel pour équipes (B2B)** — pensez Slack, Teams ou Discord, version professionnelle. Ce n'est pas un exercice jetable : vous allez le concevoir, le déployer en production dès les premières semaines, et le faire vivre comme un vrai produit, landing page et pricing compris.

Projet 1 — Messagerie temps réel (B2B)

Un vrai produit SaaS pour équipes — conçu, déployé dès les premières semaines, et vendu en fin de parcours.

25 % de l'année

MVP obligatoire

Le socle non négociable

- Conversations privées (1:1) et de groupe**
Membres, accès, historique des messages
- Temps réel**
Envoi et réception sans rechargement de page
- Notifications temps réel**
In-app au minimum : badge, toast, compteur de non-lus
- Authentification + permissions**
Qui a accès à quelle conversation ?
- Interface responsive**
Mobile, tablette, desktop
+ optionnelles valorisées : co-édition, fichiers, chatbots, recherche...

GitHub Flow

Travail en groupes de 2

```

graph LR
    A[feature/<nom>  
1 branche = 1 sujet] --> B[Pull Request  
CI : lint + build]
    B --> C[Review  
par un pair]
    C --> D[main  
= production]
    D --> E[Déploiement automatique]
  
```

Chaque merge sur **main** part en prod.
Statut de build et logs accessibles.

Au moins 1 PR significative par personne et par semaine

Repo public de préférence · rôles tournants : lead technique, QA, déploiement.

Contraintes techniques

- Backend : langage interprété uniquement
PHP, Python, JavaScript/TypeScript, Ruby...
- 1 framework backend + 1 framework frontend
- Stack validée par le professeur avant la première ligne de code
- Hébergeur : persistance + WebSockets

Remise

02/11/2026 23h59

Lien GitHub + rapport PDF dans le devoir Teams

Éliminatoire : noms des membres · format PDF

Présentation orale : un pitch business

Vous ne présentez pas un exercice, mais votre produit.

- Les fonctionnalités et le besoin auquel elles répondent
- Démo live, en production
- Justification de la stack, de l'hébergement, du temps réel
- Landing page publique + pricing d'au moins 2 paliers
- Support visuel soigné, propos structuré, temps maîtrisé

SODEV · Projet d'intégration de développement · Projet 1

7 semaines de cours · 3 revues d'avancement · présentation orale à la rentrée

MVP obligatoire

- **Conversations privées (1:1) et de groupe**
- **Temps réel** : envoi/réception sans rechargement de page
- **Notifications en temps réel** (in-app au minimum : badge, toast, compteur de non-lus)
- **Authentification + permissions basiques** (qui a accès à quelle conversation)
- **Interface responsive**

En plus de ce socle, des fonctionnalités optionnelles au choix sont fortement valorisées : co-édition de documents, base de connaissances, stockage de fichiers, chatbots, recherche plein texte... ou votre propre idée. Une optionnelle bien intégrée vaut mieux que trois optionnelles bâclées.

2. Livrables

À la remise, vous fournissez une **URL de production publique** fonctionnelle, votre **repo GitHub** (public de préférence) et deux documents : le **rapport de projet au format PDF** et un **carnet métacognitif IA par membre** (templates fournis pour les deux). La **présentation orale** devant la classe a lieu au cours suivant la remise.

Le rapport couvre notamment : les acteurs du système (rôles, permissions, interactions), le MVP et les exigences non fonctionnelles, **le choix de la stack technique et sa justification**, le schéma d'architecture, le diagramme de classes UML, la présentation de l'application avec captures d'écran, et l'explication du CI/CD et du déploiement.

⚠ Éliminatoire

- **Noms de tous les membres du groupe sur le rapport**, sinon **0**.
- **Rapport au format PDF**, sinon **0**.

3. Organisation d'équipe & Git

Vous travaillez en **groupes de 2** (fortement recommandé — les groupes de 3 restent à la discrétion du professeur), idéalement avec des rôles tournants : lead technique, référent QA, référent déploiement.

Côté Git, on applique le **GitHub Flow** : une branche `feature/<nom>` par fonctionnalité, une Pull Request revue par un pair, puis merge. `main` = production : chaque merge déclenche un déploiement. Un chapitre GitHub Flow est fourni si vous voulez apprendre à collaborer sans vous marcher sur les pieds.

Une exigence chiffrée : **au moins 1 PR significative par personne et par semaine** — substantielle (pas une micro-PR), avec des étapes de test claires.

4. Contraintes techniques

- **Backend : langage interprété uniquement** (PHP, Python, JavaScript/TypeScript, Ruby...)
- Un **framework backend** et un **framework frontend**
- **Stack validée par le professeur** avant la première ligne de code
- Secrets via variables d'environnement — jamais dans le repo

Le choix de l'hébergement guide le choix de la stack, pas l'inverse. Gardez vos exigences fonctionnelles en tête au moment de choisir : il vous faut de la **persistance des données** et du **temps réel** (WebSockets ou équivalent) — tous les hébergeurs ne s'y prêtent pas.

5. CI/CD (mis en place dès le cours 2)

Votre pipeline vérifie chaque **PR** (lint + build, tests s'ils existent) et déploie automatiquement à chaque merge sur **main**, avec un statut de build et des logs accessibles. On démarre avec cette base au deuxième cours, puis on enrichira le pipeline au fil de l'année.

6. Page marketing & pricing

Votre produit est un SaaS B2B : il lui faut une vitrine. Prévoyez une **landing page publique** qui vend les bénéfices du produit (captures animées recommandées), un **pricing d'au moins 2 paliers** (ex. Free / Pro) avec des quotas — messages, pièces jointes, membres, rétention... — et un **CTA clair** : « Essayer maintenant ».

7. Qualité & sécurité (socle)

Le lint est systématique et le formatage homogène. Des tests sur le cœur du système (envoi/lecture de message, droits d'accès) sont attendus si possible. Les entrées utilisateur sont validées et échappées, les origines CORS strictes.

Soignez particulièrement l'**expérience d'erreur** : messages d'erreur clairs dans tous les formulaires, gestion propre des exceptions, et messages compréhensibles pour prévenir l'utilisateur quand quelque chose se passe mal — jamais de stack trace à l'écran.

8. Déroulé du premier cours

1. Présentation du cours
2. Présentation du Projet 1
3. **Formation des groupes**
4. **Activité : recherche de solutions d'hébergement** — par groupe, puis mise en commun en classe ; le choix de la stack découle de vos trouvailles
5. **Début du cahier des charges** fonctionnel et non fonctionnel

9. Planning indicatif (7 semaines de cours)

Sem.	Séance	En parallèle, votre projet
S1	Présentation, groupes, activité hébergement, CDC v0	Choix stack + hébergeur à faire valider
S2	Théorie : CI/CD	Squelette du projet, auth basique, premier déploiement

Sem.	Séance	En parallèle, votre projet
S3	Revue d'avancement n°1 (rapport + démo)	DB conversations/messages, chat 1:1 temps réel
S4	Théorie	Chat de groupe, memberships, permissions
S5	Revue d'avancement n°2 (rapport + démo)	Notifications temps réel, stabilisation
S6	Théorie	Landing + pricing, fonctionnalités optionnelles
S7	Revue d'avancement n°3 + derniers conseils	Finitions, rapport PDF, démo fluide en prod
Congés	2 semaines pour finaliser → remise le 02/11 à 23h59	+ préparation de la présentation orale
Rentrée	Présentations orales devant la classe	

10. Revue d'avancement — mode d'emploi

Toutes les deux semaines, votre groupe dépose un rapport d'avancement (fonctionnalités implémentées + **captures d'écran**) dans le devoir Teams **avant la séance**, puis le présente en live : démo en production, un point fort, un point faible, et l'objectif pour la quinzaine suivante. Vous repartez avec le feedback et les conseils du professeur.

11. Présentation orale — un pitch business

Vous ne présentez pas un exercice scolaire, vous présentez **votre produit**. Le jour J, devant la classe :

- **description des fonctionnalités** — et surtout *pourquoi* elles existent (quel besoin utilisateur ?)
- **démo en live**, en production
- **justification des choix techniques** : stack, hébergement, architecture, temps réel
- support visuel soigné, propos structuré, gestion du temps

La qualité de la présentation (support et expression orale) fait partie de l'évaluation. C'est votre entraînement pour la défense du TFE.

Resources utiles

- [deck2-projet1.pdf](#)
- [grille_evaluation_Projet_1.pdf](#)
- [carnet_metacognitif_5idev.docx](#)

- [rapport_projet1_5idev.docx](#)

1.1 À la recherche de l'hébergeur idéal

Premier cours · par groupe · 20 minutes de recherche chrono + mise en commun en classe

Pourquoi cette activité ?

Vous allez déployer votre application **dès la deuxième semaine** et la garder en production pendant 7 semaines. Le choix de l'hébergement n'est pas un détail de fin de projet : c'est lui qui va **conditionner votre choix de stack** — pas l'inverse.

Un hébergeur génial qui ne supporte pas les WebSockets ? Éliminé. Un free tier qui efface votre base de données toutes les nuits ? Éliminé (ou à connaître !). C'est maintenant qu'on le découvre, pas en semaine 5.

Consignes

1. **En groupe**, identifiez et analysez **au moins 3 solutions d'hébergement** (gratuites ou payantes).
2. Remplissez la **grille d'analyse** ci-dessous pour chacune. Notez systématiquement les **prix et les quotas offerts** par chaque solution — c'est ce qui permettra de comparer en classe.
3. Désignez votre **candidat favori** et préparez 2 minutes de pitch : pourquoi lui ?
4. **Mise en commun en classe** : chaque groupe présente ses trouvailles, on construit ensemble la carte des options.
5. À l'issue de l'activité : proposez votre **duo hébergeur + stack** au professeur pour validation.

Points d'attention

Votre projet est une **messagerie temps réel**. Vérifiez impérativement :

- **Persistance des données** : la base de données survit-elle aux redéploiements ? Aux périodes d'inactivité ?
- **Temps réel** : WebSockets supportés ? Connexions longues autorisées ? Ou faut-il un service tiers ?
- **Mise en veille** : certains free tiers endorment l'app après X minutes d'inactivité — quel impact pour votre démo ?

Rappel : backend en **langage interprété** uniquement (PHP, Python, JS/TS, Ruby...).

Grille d'analyse (à remplir par solution)

Solution : _____

Critère	Vos observations
Type de service (PaaS, VPS, conteneurs, serverless...)	
Coût : free tier ? assez généreux pour tenir 7 semaines ? prix et quotas des paliers ? prix si dépassement ? carte de crédit exigée ?	
Langages/runtimes supportés (lesquels sont interprétés ?)	
Base de données : proposée ? persistante ? quelle techno ? quelles limites (taille, connexions) ?	
Temps réel : WebSockets OK ? durée max des connexions ?	
Mise en veille / cold start : l'app s'endort-elle ? délai de réveil ?	
Déploiement : depuis GitHub ? déploiement auto sur push ?	
Variables d'environnement / secrets : gérés comment ?	
Logs & monitoring : accessibles ?	
Limites diverses : bande passante, builds/mois, nombre d'apps, domaine custom ?	

Verdict du groupe

Question	Réponse
Notre favori et pourquoi (3 arguments max)	
Ce qui nous inquiète / à surveiller	
Stack envisagée en conséquence (backend + frontend + DB)	
Plan B si le favori nous lâche	

Livrable

La grille remplie (photo, PDF ou markdown) déposée sur Teams en fin de séance + votre proposition **hébergeur + stack** pour validation.

1.2 Workflows de collaboration avancés

Modèles de branchement : Comprendre les stratégies

La collaboration sur un projet Git nécessite une approche structurée de la gestion des branches. Deux modèles principaux émergent : **Git Flow** et **GitHub Flow**, chacun avec ses propres forces et cas d'utilisation.

Git Flow : Un modèle complet pour les projets complexes

Git Flow est un modèle de branchement robuste, particulièrement adapté aux projets avec des cycles de release bien définis.

Structure typique des branches :

- `main` (ou `master`) : branche de production
- `develop` : branche d'intégration pour le développement
- `feature/*` : branches pour les nouvelles fonctionnalités
- `release/*` : branches de préparation de release
- `hotfix/*` : branches pour les corrections critiques

Exemple de workflow Git Flow :

1. Créer une branche `feature` depuis `develop`
2. Développer la fonctionnalité
3. Merger la `feature` dans `develop`
4. Créer une branche `release` pour les tests
5. Merger `release` dans `main` et `develop`
6. Appliquer des tags de version

GitHub Flow : Approche simplifiée et continue

GitHub Flow privilégie la simplicité et le déploiement continu.

Structure des branches :

- `main` : branche de production toujours déployable
- `feature/*` : branches pour chaque développement

Processus :

1. Créer une branche à partir de `main`
2. Développer et commiter
3. Ouvrir une Pull Request

4. Revue et discussion
5. Déployer depuis la branche
6. Merger dans `main`

Comparaison des approches

Critère	Git Flow	GitHub Flow
Complexité	Élevée	Faible
Releases planifiées	Oui	Non
Déploiement continu	Difficile	Facile
Adapté à	Logiciels avec cycles	Développement web, startups

Choix du workflow : Considérations pratiques

Le choix entre Git Flow et GitHub Flow dépend de plusieurs facteurs :

- Nature du projet
- Fréquence des releases
- Taille de l'équipe
- Processus de déploiement

Conseils de sélection :

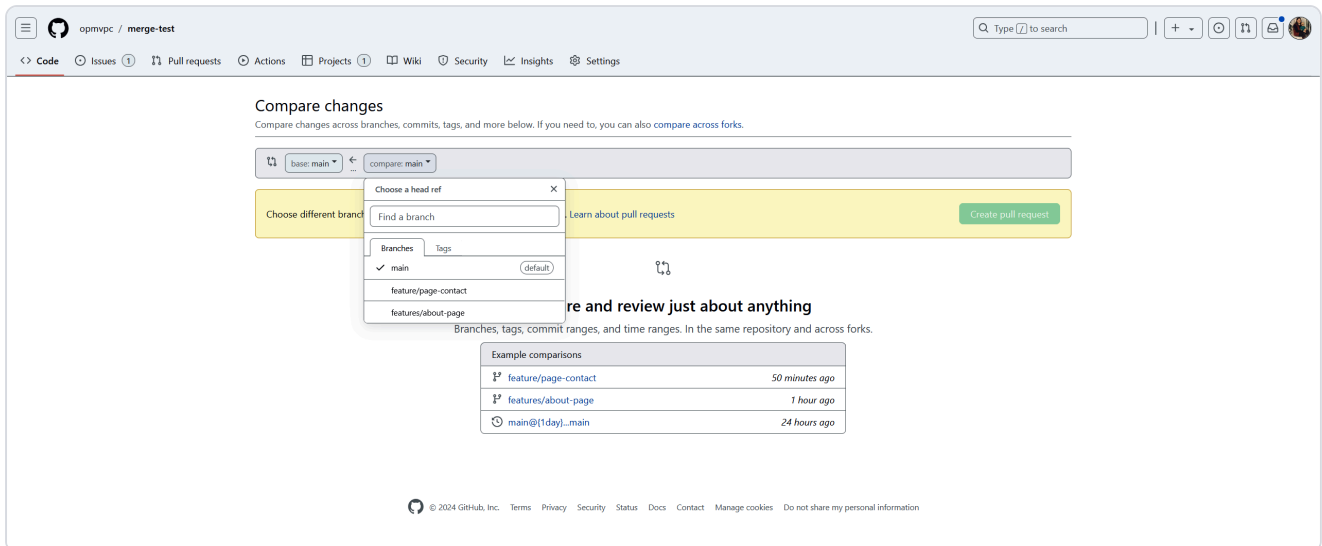
- Projets avec versions stables → **Git Flow**
- Développement web, MVP → **GitHub Flow**
- Projets open-source → Généralement **GitHub Flow**

Pull Request : Concept et Définition

Qu'est-ce qu'une Pull Request ?

Une Pull Request (PR) est bien plus qu'un simple mécanisme technique de fusion de code. C'est un outil de communication et de collaboration qui permet aux développeurs de proposer des modifications à un projet partagé. Imaginez-la comme une invitation à discuter et à examiner ensemble une série de changements avant de les intégrer définitivement au projet.

Dans le workflow GitHub, une Pull Request signale à l'équipe : "J'ai travaillé sur ces modifications. Pouvez-vous les examiner et me donner votre avis avant que je ne les incorpore au projet principal ?" C'est un moment crucial d'échange et de validation collective.



Processus de base d'une Pull Request

La création d'une Pull Request suit généralement ces étapes :

1. Créer une nouvelle branche

- Partez toujours de la branche principale (généralement `main` ou `master`)
- Donnez un nom clair et descriptif à votre branche (ex: `feature/login-authentication`)

2. Développer votre fonctionnalité ou correction

- Travaillez sur votre branche
- Effectuez des commits réguliers et significatifs
- Assurez-vous que votre code fonctionne localement

3. Pousser votre branche sur GitHub

```
git push -u origin feature/login-authentication
```

BASH

4. Ouvrir la Pull Request

- Sur GitHub, sélectionnez votre branche
- Cliquez sur "New Pull Request"
- Décrivez vos modifications
- Indiquez le contexte et les raisons de ces changements

5. Revue et discussion

- Les membres de l'équipe examinent votre code
- Ils peuvent commenter, suggérer des modifications
- Vous pouvez répondre, clarifier, modifier votre code

6. Finalisation

- Si tout est validé : merger la branche
- Si des modifications sont nécessaires : les apporter et mettre à jour la PR

Code Review : Concepts et Pratiques

Définition de la Code Review

La code review est un processus collaboratif essentiel dans le développement logiciel moderne. Il s'agit d'un examen systématique du code écrit par un développeur, effectué par ses pairs ou des membres plus expérimentés de l'équipe.

Les objectifs principaux sont multiples :

- Détecter et corriger les bugs potentiels
- Partager les connaissances au sein de l'équipe
- Maintenir une haute qualité de code
- Garantir la cohérence technique du projet
- Faciliter l'apprentissage collectif

Risques d'une absence de Code Review

Sans revue de code, une équipe s'expose à plusieurs risques significatifs :

- Bugs non détectés qui peuvent s'accumuler
- Code peu lisible et difficilement maintenable
- Isolation des connaissances techniques
- Accumulation rapide de dette technique
- Failles de sécurité potentiellement non identifiées
- Divergence des pratiques de codage au sein de l'équipe

Bonnes Pratiques de Code Review

Qui peut faire une revue ?

Une revue de code efficace implique généralement :

- Les développeurs expérimentés du projet
- Les membres directs de l'équipe
- Le lead technique ou l'architecte
- Les contributeurs réguliers ayant une bonne compréhension du projet

L'idéal est de varier les reviewers pour bénéficier de perspectives différentes.

Comment bien faire une revue

Critères d'évaluation :

- Lisibilité et compréhension du code
- Respect des conventions du projet
- Logique et efficacité de l'implémentation
- Performance et complexité algorithmique
- Gestion appropriée des erreurs
- Qualité et pertinence des commentaires
- Couverture et qualité des tests

Approche constructive :

- Commenter le code, jamais la personne
- Être précis et bienveillant
- Proposer des alternatives concrètes
- Expliquer le raisonnement derrière chaque suggestion

Exemple de commentaire constructif :

Cette méthode pourrait bénéficier d'une refactorisation.
Je suggère d'extraire la logique de validation dans une méthode dédiée,
ce qui améliorerait la lisibilité et permettrait de réutiliser ce comportement.

Intégration Continue avec Laravel et GitHub Actions

Définition

L'intégration continue est une pratique de développement qui consiste à :

- Automatiser la compilation et les tests du code
- Détecter rapidement les problèmes
- Garantir la qualité à chaque modification

Configuration pour un projet Laravel

Voici un exemple de workflow GitHub Actions adapté à Laravel :

YAML

```

name: Laravel Tests
on: [push, pull_request]
jobs:
  tests:
    runs-on: ubuntu-latest
    steps:
      - uses: actions/checkout@v2
      - name: Setup PHP
        uses: shivammathur/setup-php@v2
        with:
          php-version: '8.3'

      - name: Install Dependencies
        run: composer install -q --no-ansi --no-interaction --no-scripts --no-progress --prefer-dist

      - name: Generate key
        run: php artisan key:generate

      - name: Run Pest
        run: ./vendor/bin/pest

      - name: Run PHPStan
        run: ./vendor/bin/phpstan analyse

```

Installation des outils

BASH

```

# Installation de Pest
composer require pestphp/pest --dev --with-all-dependencies

# Installation de PHPStan
composer require --dev phpstan/phpstan

```

Exemple de configuration PHPStan (`phpstan.neon`) :

```

parameters:
  level: 6
  paths:
    - app
    - tests

```

Cette configuration garantira que chaque modification passe les tests et respecte les standards de qualité de code.

Techniques de Travail Collaboratif : Résolution des Conflits

Comprendre les Conflits de Fusion

Qu'est-ce qu'un Conflit de Fusion ?

Un conflit de fusion survient lorsque deux développeurs modifient les mêmes parties d'un fichier dans différentes branches. Git, malgré sa sophistication, ne peut pas toujours décider automatiquement quelle version conserver.

Imaginez deux développeurs travaillant simultanément sur le même fichier PHP. L'un modifie une méthode de validation, l'autre change la logique de cette même méthode. Lorsqu'ils tentent de fusionner leurs branches, Git signale un conflit, nécessitant une intervention manuelle.

Types de Conflits

Il existe principalement deux catégories de conflits :

1. **Conflits de Contenu** : Ils surviennent quand les mêmes lignes de code sont modifiées différemment. Par exemple :

```
// Version sur ma branche
public function validate(data) {
    return $data['email'] === '';
}

// Version sur la branche principale
public function validate(data) {
    return filter_var($data['email'], FILTER_VALIDATE_EMAIL);
}
```

PHP

2. **Conflits Structurels** : Ils concernent des changements plus globaux comme la réorganisation du code, le déplacement de méthodes ou la modification de l'architecture.

Prévention des Conflits

La meilleure stratégie pour gérer les conflits est de les prévenir :

Communication d'Équipe

- Informez vos collègues de vos développements en cours
- Utilisez les issues GitHub pour planifier le travail
- Synchronisez régulièrement votre branche avec la branche principale

Pratiques de Développement

- Créez des branches courtes et ciblées
- Mergez fréquemment
- Évitez de travailler trop longtemps sur des branches isolées

Processus de Résolution de Conflits

Identification des Conflits

Première étape : identifier les fichiers problématiques

```
git status
```

BASH

Cette commande affichera les fichiers en conflit.

Comprendre la Structure d'un Conflit

Git marque les zones conflictuelles avec des balises spécifiques :

```
<<<<<<< HEAD
// Votre version actuelle
public function valider(data) {
    return $data['email'] !== '';
}
=====
// Version en cours de fusion
public function valider(data) {
    return filter_var($data['email'], FILTER_VALIDATE_EMAIL);
}
>>>>>>> nouvelle_branche
```

PHP

Résolution Manuelle

Étapes de résolution :

1. Ouvrez le fichier en conflit
2. Éditez manuellement pour garder la meilleure implémentation
3. Supprimez les balises de Git
4. Choisissez l'implémentation la plus appropriée

Exemple de résolution :

PHP

```
public function validate($data) {  
    return !empty($data['email']) &&  
        filter_var($data['email'], FILTER_VALIDATE_EMAIL);  
}
```

Validation de la Résolution

Après avoir résolu manuellement :

BASH

```
git add fichier_resolu  
git commit -m "Résolution du conflit de fusion"
```

Outils de Résolution dans Visual Studio Code

VSCode offre des outils visuels puissants pour résoudre les conflits :

- Boutons interactifs de résolution
- Vue comparative côte à côte
- Sélection facilitée des modifications

Stratégies de Choix

Lors de la résolution, considérez :

- La logique métier
- La performance du code
- La cohérence avec l'architecture existante
- L'intention originale du développement

Conseils Éthiques et Pratiques

Lors d'un Conflit

- Communiquez toujours avec le développeur concerné
- Cherchez à comprendre le contexte des modifications
- Privilégiez la solution technique la plus propre
- Restez professionnel et constructif

Cette approche transforme un conflit potentiellement frustrant en une opportunité de collaboration et d'amélioration du code.

GitHub Projects : Outil de Suivi et de Planification

Présentation de GitHub Projects

GitHub Projects est un outil intégré de gestion de projet directement dans l'écosystème GitHub. Il permet aux équipes de développement de suivre, organiser et planifier leur travail de manière visuelle et collaborative.

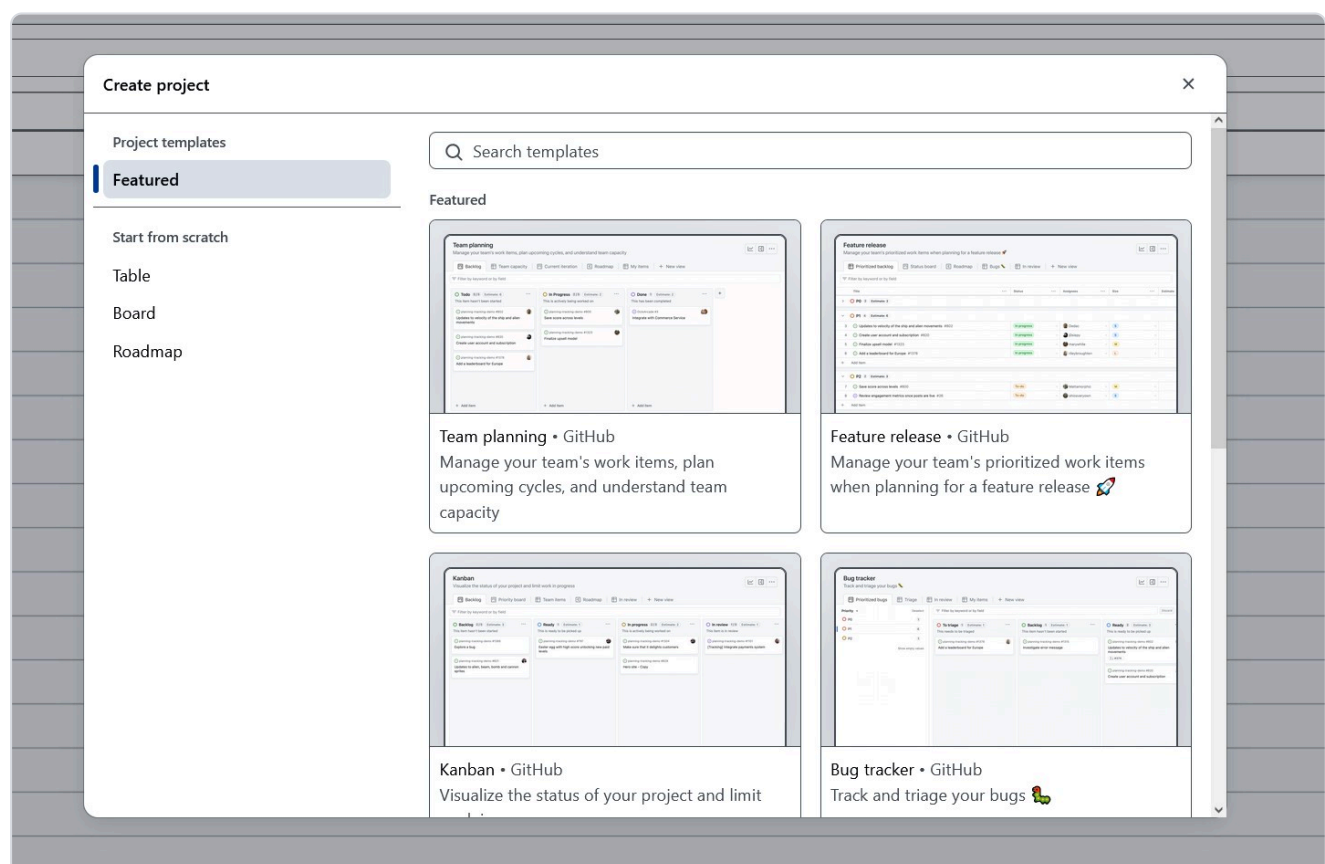
Les projets GitHub peuvent prendre plusieurs formes :

- Tableau Kanban : colonnes représentant les étapes de progression
- Liste : vue linéaire des tâches
- Tableau de bord : vue personnalisée selon les besoins de l'équipe

Création d'un Projet

La création d'un projet suit quelques étapes simples :

1. Dans votre dépôt GitHub, cliquez sur "Projects"
2. Choisissez "New Project"
3. Sélectionnez un modèle de base ou commencez à zéro



Configuration typique des colonnes :

- "À faire" : Tâches à planifier
- "En cours" : Développements actifs
- "Terminé" : Tâches réalisées

Exemple visuel des colonnes :

@opmvpc's untitled project

BacklogTeam capacityCurrent iterationRoadmapMy items+ New view

Filter by keyword or by field

Todo2 / 5Estimate: 0

This item hasn't been started

Draft

Ajouter Tests

Draft

Ajouter GitHub Action

+ Add item

In Progress1 / 5Estimate: 0

This is actively being worked on

merge-test #3

Ajouter page About

+ Add item

Done1Estimate: 0

This has been completed

Draft

Ajouter page Contact

+ Add item

Modification d'une tâche

Ajouter page Contact

Draft
opmvpc opened 2 minutes ago

Edit

Assignees
opmvpc

Status
Done

Priority
P0

Size
S

Estimate
Enter a number...

Iteration
Iteration

Start date
Nov 27, 2024

End date
Nov 27, 2024

Convert to issue

Copy link in project

Archive

Delete from project

opmvpc now (edited)

Edit

No description provided

Issues GitHub : Gestion des Tâches et Bugs

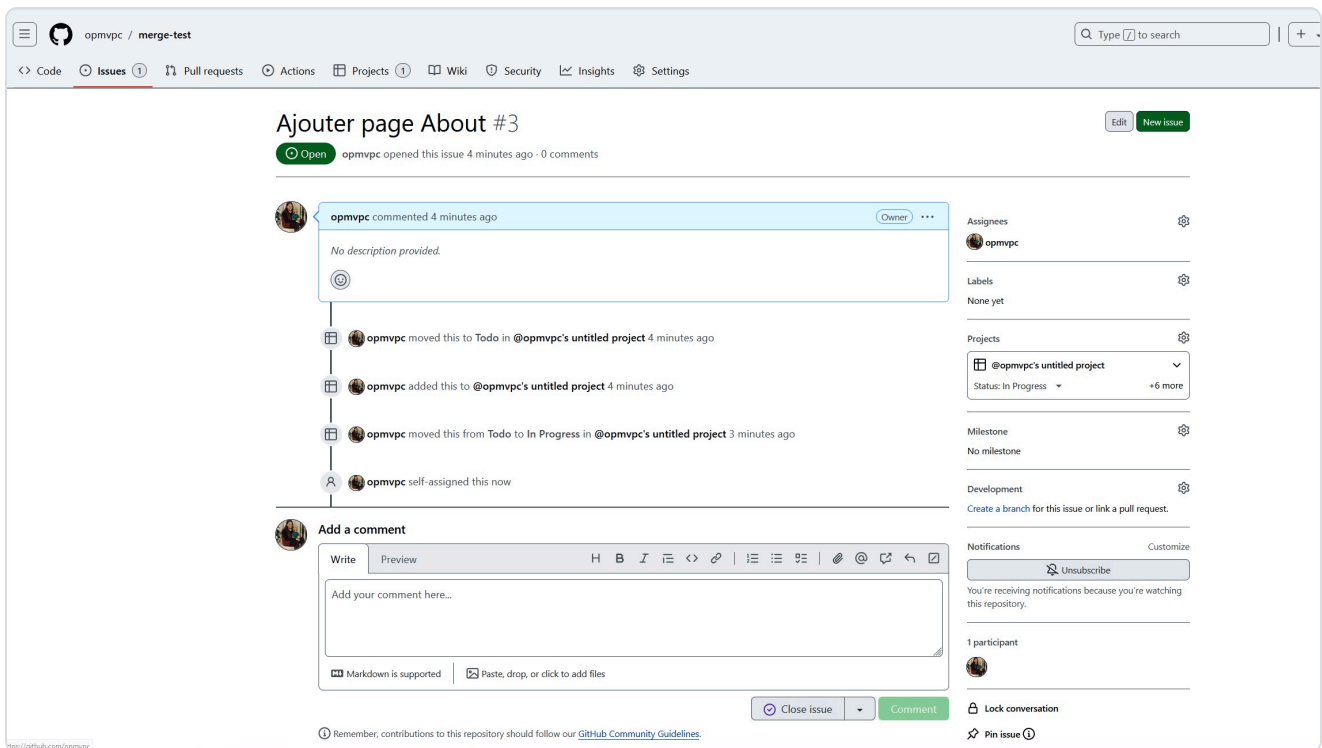
Qu'est-ce qu'une Issue ?

Une issue GitHub est un outil de communication et de suivi dans un projet. Elle permet de :

- Signaler un bug
- Proposer une amélioration
- Suivre une tâche à réaliser
- Discuter d'un point technique

Types d'issues courants :

- Bug : Problème technique à résoudre
- Amélioration : Nouvelle fonctionnalité
- Tâche : Travail à accomplir
- Documentation : Mise à jour de la documentation



Création et Gestion des Issues

Informations importantes lors de la création :

- Titre clair et concis
- Description détaillée
- Contexte du problème
- Étapes de reproduction (pour les bugs)
- Résultat attendu

Exemple d'une bonne issue :

YAML

Titre: Correction de la validation d'email dans le formulaire d'inscription

Description:

Actuellement, le formulaire accepte des emails invalides.

Étapes de reproduction :

1. Aller sur la page d'inscription
2. Entrer un email sans format valide
3. Soumettre le formulaire

Résultat attendu :

Le formulaire doit rejeter les emails mal formatés

Gestion de l'issue :

- Assignment à un développeur
- Définition d'une priorité
- Suivi de son avancement

Étiquettes (Labels)

Rôle des Labels

Les labels sont des étiquettes colorées qui permettent de :

- Catégoriser rapidement une issue
- Communiquer visuellement son statut
- Organiser les tâches

Types de Labels Courants

Labels de priorité :

- `priorité: haute`
- `priorité: moyenne`
- `priorité: basse`

Labels de type :

- `bug`
- `amélioration`
- `documentation`
- `question`

Labels de statut :

- `en cours`
- `bloqué`
- `nécessite des informations`

Création et Utilisation des Labels

Bonnes pratiques :

- Utilisez des couleurs distinctives
- Créez des labels personnalisés adaptés à votre projet
- Soyez cohérent dans leur utilisation
- Limitez le nombre de labels pour éviter la confusion

Exemple de configuration de labels :

- bug (couleur rouge)
- amélioration (couleur jaune)
- documentation (couleur verte)

Exercice pratique : gestion collaborative de projets avec Git et Laravel

Introduction à l'exercice

Dans cet exercice, vous travaillerez par groupes de deux pour développer une application Laravel utilisant Breeze. Vous apprendrez à gérer des branches Git, à écrire des tests HTTP avec Pest, à créer et résoudre des conflits Git, et à configurer l'intégration continue avec GitHub Actions.

Mise en place du projet

1. Initialisation du dépôt Git :

- Un étudiant crée un nouveau projet Laravel
- Ensuite, il le push sur GitHub
- L'autre étudiant clone le dépôt localement.

Création des fonctionnalités

1. Création de branches :

- Chaque étudiant crée une branche à partir de `main`, par exemple, `feature/about-page` et `feature/contact-page`.

2. Développement de nouvelles pages et contrôleurs :

- **Étudiant A** : Crée une page "About" avec un contrôleur `PageController`.

```
// web.php
Route::get('/about', [PageController::class, 'showAbout']);

// PageController.php
public function showAbout()
{
    return view('about');
}
```

PHP

- **Étudiant B** : Crée une page "Contact" avec le même contrôleur `PageController`.

PHP

```
// web.php
Route::get('/contact', [PageController::class, 'showContact']);

// PageController.php
public function showContact()
{
    return view('contact');
}
```

Simulation et résolution de conflits

1. Création de Pull Requests :

- Chaque étudiant soumet une Pull Request de sa branche vers `main`.

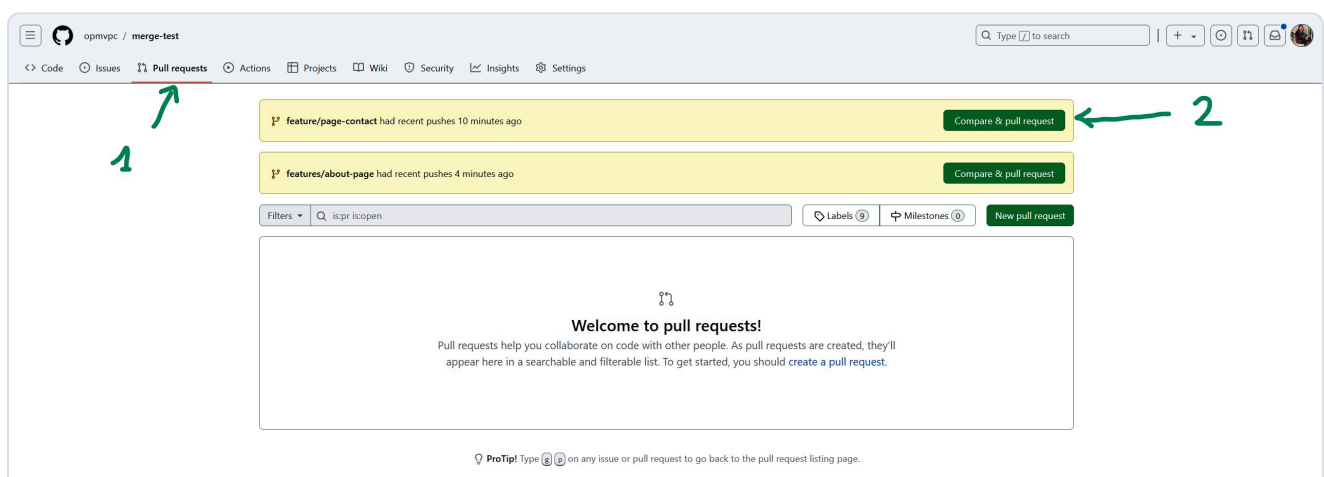
2. Scénarios de conflits :

- Les deux étudiants ont modifié les fichiers `web.php` et `PageController.php`, ce qui crée des conflits lors du merge des Pull Requests.

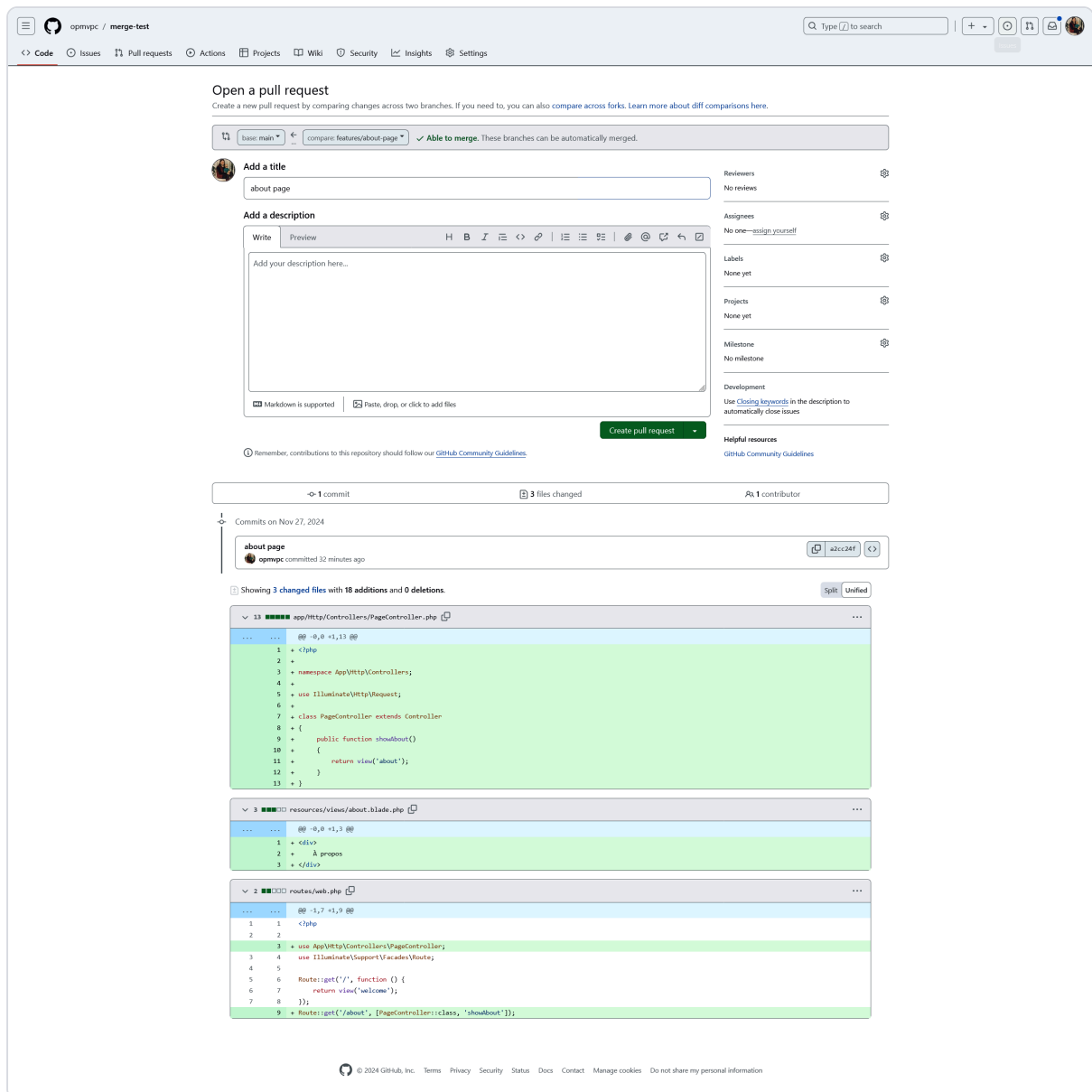
3. Processus de résolution de conflits :

- Ouvrez les fichiers en conflit.
- Discutez des modifications à conserver et résolvez les conflits en intégrant les routes et méthodes de chaque branche.
- Validez les modifications et complétez le merge.

Création d'une Pull Request



Créer la première Pull Request



Validez les changements et lancez le Merge

Créez la 2e Pull Request et essayez de Merge

Open a pull request
Create a new pull request by comparing changes across two branches. If you need to, you can also [compare across forks](#). [Learn more about diff comparisons here](#).

base: main ← compare: feature/page-contact × Can't automatically merge. Don't worry, you can still create the pull request.

Add a title
install breeze+page contact

Add a description
Write Preview H B I

Add your description here...

Markdown is supported Paste, drop, or click to add files

Create pull request

Remember, contributions to this repository should follow our [GitHub Community Guidelines](#).

Reviewers
No reviews

Assignees
No one—assign yourself

Labels
None yet

Projects
None yet

Milestone
No milestone

Development
Use [Closing keywords](#) in the description to automatically close issues

Helpful resources
[GitHub Community Guidelines](#)

Écriture des tests avec Pest

1. Installez Breeze avec l'option Pest pour les tests.

2. Création de tests HTTP :

○ Test de la page About :

```
test('la page About se charge correctement', function () {
    response=response = response=this->get('/about');
    response->assertStatus(200);response->assertStatus(200);
    response->assertStatus(200);response->assertSee('À propos');
});
```

PHP

○ Test de la page Contact :

```
test('la page Contact se charge correctement', function () {
    response=response = response=this->get('/contact');
    response->assertStatus(200);response->assertStatus(200);
    response->assertStatus(200);response->assertSee('Contactez-nous');
});
```

PHP

3. Lancez les tests avec la commande `php artisan test` et corrigez votre code si besoin.

Intégration continue avec GitHub Actions

1. Configuration du workflow :

- Créez un fichier `.github/workflows/laravel.yml` avec le contenu suivant :

YAML

```

name: Run Laravel Tests

on: [push, pull_request]

jobs:
  tests:
    runs-on: ubuntu-latest
    steps:
      - uses: actions/checkout@v2

      - name: Setup PHP
        uses: shivammathur/setup-php@v2
        with:
          php-version: "8.3"

      - name: Install Dependencies
        run: composer install -q --no-ansi --no-interaction --no-scripts --no-progress --prefer-dist

      - name: Setup Node
        uses: actions/setup-node@v2
        with:
          node-version: "22"

      - name: Install NPM Dependencies
        run: npm install

      - name: Build Assets
        run: npm run build

      - name: Copy .env
        run: php -r "file_exists('.env') || copy('.env.example', '.env');"

      - name: Install Dependencies
        run: composer install -q --no-ansi --no-interaction --no-scripts --no-progress --prefer-dist

      - name: Generate key
        run: php artisan key:generate

      - name: Directory Permissions
        run: chmod -R 777 storage bootstrap/cache

      - name: Create Database
        run: |
          mkdir -p database
          touch database/database.sqlite
          php artisan migrate --force

      - name: Execute tests (Unit and Feature tests) via PHPUnit/Pest
        env:
          DB_CONNECTION: sqlite
          DB_DATABASE: database/database.sqlite
        run: php artisan test

```

2. Fonctionnement du workflow :

- Le workflow s'exécute à chaque push ou pull request, vérifiant que les tests passent correctement.

3. Pushez votre code et voyez si les tests passent.

1.3 CI/CD : votre filet de sécurité automatique

Point de théorie · semaine 2 · à mettre en place sur votre Projet 1 avant la revue d'avancement n°1

Slides du cours

-  [deck3-cicd.pdf](#)  — les 16 slides du point de théorie

Vous savez écrire du code. Ce chapitre vous apprend à laisser un robot le vérifier à chaque Pull Request et à le livrer en production à chaque merge, sans que personne n'ait à y penser. C'est la brique d'industrialisation la plus rentable que vous apprendrez cette année : une heure de mise en place, sept semaines de nuits tranquilles.

À la fin de ce chapitre, vous serez capables de :

1. **expliquer** ce qu'un pipeline CI/CD fait, quand il s'exécute et sur quelle machine ;
2. **écrire** des tests unitaires et d'intégration minimaux dans votre stack (Laravel, Django ou Node) ;
3. **configurer** un workflow GitHub Actions qui lance ces tests à chaque PR, avec ou sans base de données ;
4. **brancher** le déploiement automatique de votre hébergeur sur `main`, protégé par la CI ;
5. **diagnostiquer** un run rouge sans paniquer.

Tout le code de ce chapitre est dans le repo d'exemple <https://github.com/opmvpc/cicd26> : trois mini-API de messagerie identiques (Laravel, Django, Node), chacune avec ses tests et son workflow. Clonez-le, cassez-le, regardez ce qui se passe.

1. Le problème qu'on résout

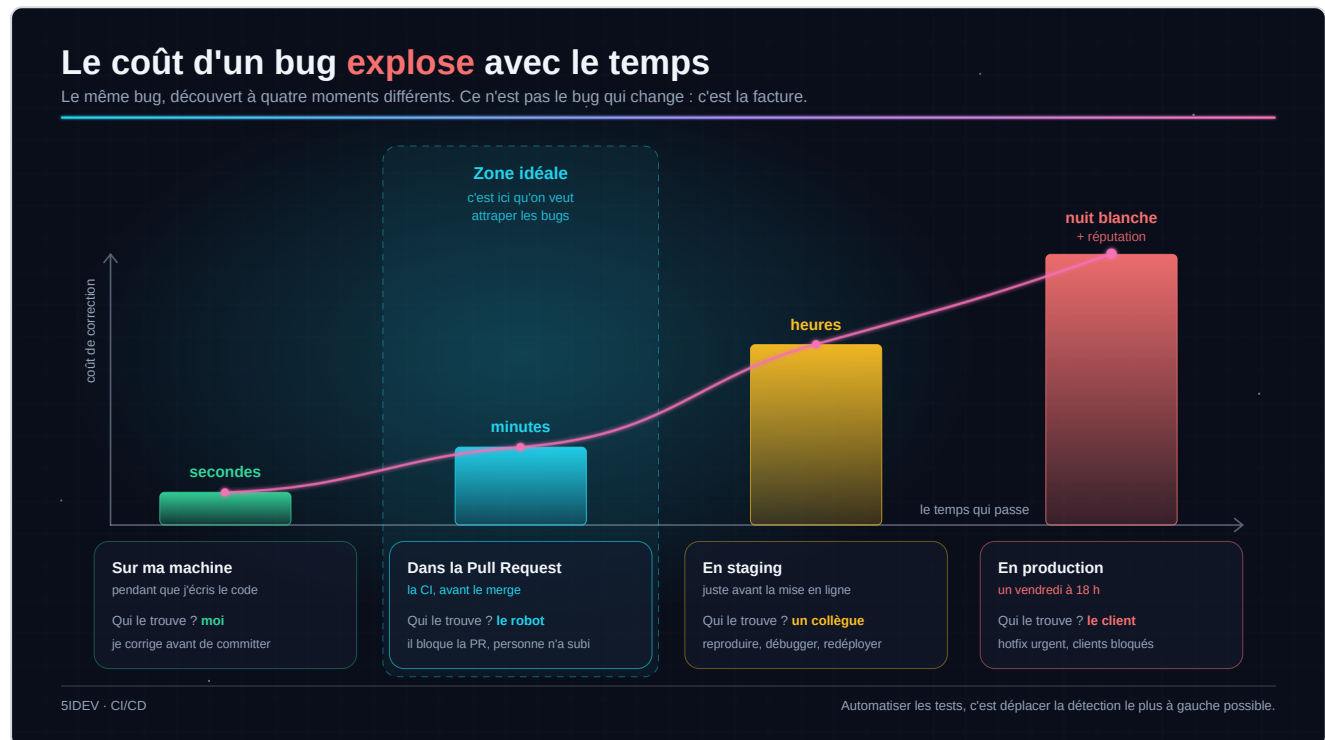
« Ça marche sur ma machine »

Vendredi, 17h50. Votre binôme merge sa branche `feature/notifications` sur `main`. Sur son PC, tout fonctionne. Sauf qu'il a renommé une colonne en base sans mettre à jour la requête qui liste les messages, et que cette page-là, il ne l'a pas rouverte depuis mardi. `main` = production :

l'hébergeur déploie. À 18h02, votre chat affiche une erreur 500 à quiconque ouvre une conversation. Vous le découvrez dimanche soir, en préparant votre démo de la revue d'avancement.

Ce scénario n'est pas une fiction : c'est la vie de tous les projets sans garde-fou. Le problème n'est pas que votre binôme est mauvais. Le problème est qu'un humain ne peut pas retester toute l'application à chaque modification. Un robot, si.

Plus un bug est trouvé tard, plus il coûte cher



Où le bug est détecté	Qui le trouve	Ce que ça coûte
Sur votre machine, en codant	Vous	Quelques secondes, personne ne le saura
Dans la Pull Request, par la CI	Le robot	Quelques minutes : vous corrigez avant le merge
En production	Un utilisateur, ou le prof pendant la démo	Une soirée, un rollback dans l'urgence, une réputation

La zone qu'on veut viser, c'est la deuxième ligne. C'est tout l'objet de l'intégration continue : attraper les bugs **dans la PR**, avant qu'ils touchent **main**.

Trois lettres, trois niveaux d'automatisation

- **CI, intégration continue** : à chaque push, une machine neutre récupère votre code, l'installe de zéro, lance le lint et les tests, et affiche vert ou rouge. « Intégration » parce qu'on vérifie que votre travail s'intègre au reste sans rien casser.
- **CD, livraison continue** (*delivery*) : si la CI est verte, une version déployable est préparée automatiquement. Un humain décide encore quand la mettre en production.
- **CD, déploiement continu** (*deployment*) : plus d'humain dans la boucle. Vert sur `main` = en production dans les minutes qui suivent.

Dans ce cours, `main` est votre production et chaque merge déploie : vous faites du **déploiement continu**. Ça n'a rien d'effrayant si la CI est solide et si personne ne peut merger tant qu'elle est rouge. On y revient en partie 5.

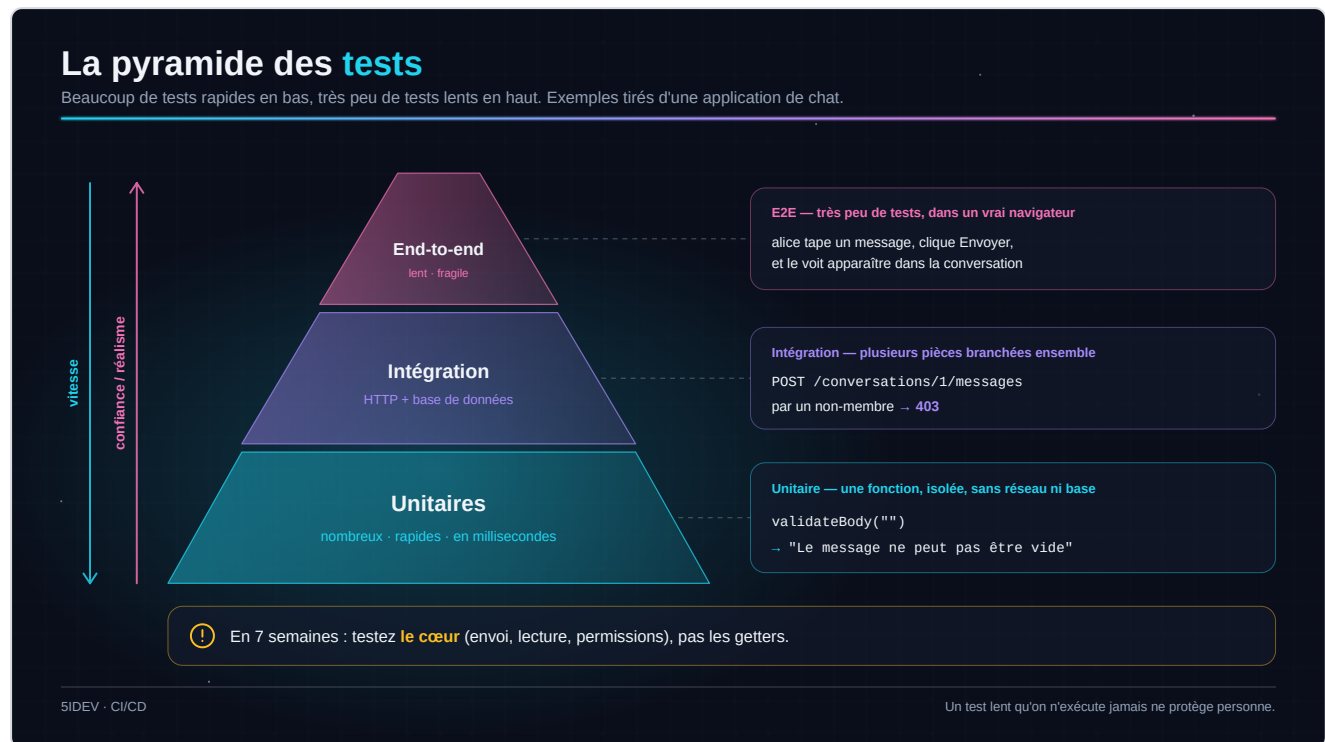
NOTE

Le mot **pipeline** désigne l'enchaînement complet : push → lint → tests → build → déploiement. Chaque étape est un tuyau qui ne laisse passer que ce qui est correct.

2. Les tests, concrètement

Un pipeline sans tests vérifie que votre code s'installe et compile. C'est déjà utile, mais le vrai filet de sécurité, ce sont les tests. Pas besoin d'en écrire des centaines : quelques tests bien placés sur le cœur de votre application suffisent à éviter la catastrophe du vendredi.

La pyramide des tests



Niveau	Ce qu'on teste	Vitesse	Exemple dans votre chat
Unitaire	Une fonction ou une classe isolée, sans HTTP ni base de données	Millisecondes	<code>validateBody(" ")</code> renvoie « Le message ne peut pas être vide »
Intégration	Plusieurs couches ensemble : requête HTTP → contrôleur → base de données → réponse	Dizaines de ms à secondes	<code>POST /conversations/1/messages</code> par un non-membre renvoie 403
End-to-end (E2E)	L'application complète pilotée par un vrai navigateur, comme un humain	Secondes à minutes	Alice tape un message, clique « Envoyer », le message apparaît dans la liste

Plus on monte, plus le test est réaliste et rassurant, mais plus il est lent, fragile et coûteux à maintenir. D'où la forme de pyramide : beaucoup d'unitaires, quelques tests d'intégration sur les routes critiques, une poignée d'E2E sur les parcours vitaux.

Quoi tester quand on a sept semaines ?

Le cœur du système, celui dont la panne vous ferait perdre des points à la démo :

- **envoyer un message** dans une conversation dont on est membre ;
- **lire les messages** d'une conversation ;

- **les permissions** : un non-membre ne lit pas et n'écrit pas ;
- **la validation** : message vide, trop long, mauvais type.

Ne testez pas les getters, les setters, ni ce que le framework teste déjà pour vous. Un test doit **casser quand vous cassez quelque chose d'important**, c'est son seul rôle.

À quoi ressemble un test dans votre stack

Le repo `cicd26` implémente exactement la même règle métier trois fois : seul un membre d'une conversation peut y lire ou écrire, et un message ne peut pas être vide. Voici le même test d'intégration dans les trois stacks. Remarquez la structure identique : préparer, agir, vérifier.

Laravel avec Pest (`tests/Feature/MessagesApiTest.php`)

PHP

```
uses(RefreshDatabase::class); // base remise à zéro entre chaque test

beforeEach(function () {
    this->conversation=Conversation::create(['name'⇒'ge'ne'ral']);this->conversation
= Conversation::create(['name' ⇒ 'général']);
    this->conversation=Conversation::create(['name'⇒'ge'ne'ral']);this->conversation-
>addMember('alice');
});

it('refuse un non-membre (403)', function () {
    $this->withHeader('X-User', 'mallory')
        ->postJson("/conversations/{this->conversation->id}/messages", ['body' ⇒
'Coucou'])
        ->assertForbidden();
});
```

Django avec pytest (`chat/tests/test_api.py`)

PYTHON

```
pytestmark = pytest.mark.django_db # chaque test tourne dans une transaction annulée
à la fin

@pytest.fixture
def conversation():
    conv = Conversation.objects.create(name="général")
    conv.members.add(Member.objects.create(username="alice"))
    return conv

def test_un_non_membre_est_refuse(client, conversation):
    res = client.post(f"/conversations/{conversation.id}/messages", {"body":
"Coucou"},
                    content_type="application/json", headers={"X-User": "mallory"})
    assert res.status_code == 403
```

Node/Express avec Vitest et supertest (tests/api.integration.test.js)

JAVASCRIPT

```

beforeEach(async () => {
  app = createApp(createDb(":memory:")); // SQLite en mémoire : base neuve à chaque
  test
  await request(app).post("/conversations").send({ name: "général", members: ["alice"]
});
});

it("un non-membre est refusé (403)", async () => {
  const res = await request(app)
    .post("/conversations/1/messages")
    .set("X-User", "mallory")
    .send({ body: "Coucou" });
  expect(res.status).toBe(403);
});

```

Et la commande qui lance tout, celle que la CI exécutera aussi :

Stack	Commande
Laravel	php artisan test
Django	pytest
Node	npm test (qui appelle vitest run)

ASTUCE

Un test isolé est un test qui repart d'une base vide. RefreshDatabase en Laravel, django_db en Django, une base SQLite :memory: en Node : c'est ce qui garantit que vos tests passent dans n'importe quel ordre, sur n'importe quelle machine, y compris celle de la CI.

Le lint et l'analyse statique : des tests que vous n'écrivez pas

Vos tests vérifient le comportement. Deux autres familles d'outils vérifient le **code lui-même**, sans l'exécuter :

- **Le lint et le formatage** vérifient le style : indentation, imports inutilisés, conventions de nommage. Pint pour Laravel, ruff pour Python, ESLint et Prettier pour JavaScript.
- **L'analyse statique** lit votre code comme un relecteur très pointilleux et y trouve des erreurs *avant l'exécution* : appel d'une méthode qui n'existe pas, variable qui peut être null là où vous la déréférenciez, type incompatible passé à une fonction. Elle ne remplace pas les tests, elle attrape une autre classe de bugs, gratuitement, sur tout le code, y compris celui que vous n'avez pas testé.

Stack	Formatage / lint	Analyse statique
Laravel	Pint	PHPStan avec Larastan
Django	ruff (lint et format)	mypy, ou pyright
Node / TypeScript	ESLint, Prettier	TypeScript (<code>tsc --noEmit</code>)

Ces outils tournent en quelques secondes et se placent dans un job à part de votre pipeline, en parallèle des tests. On les mettra en place proprement et on parlera de niveaux de sévérité, de règles et de configuration dans un prochain sous-chapitre. Pour l'instant, retenez qu'un job de lint est la première chose que vous ajouterez après vos tests.

3. GitHub Actions, anatomie

Ce qui se passe physiquement



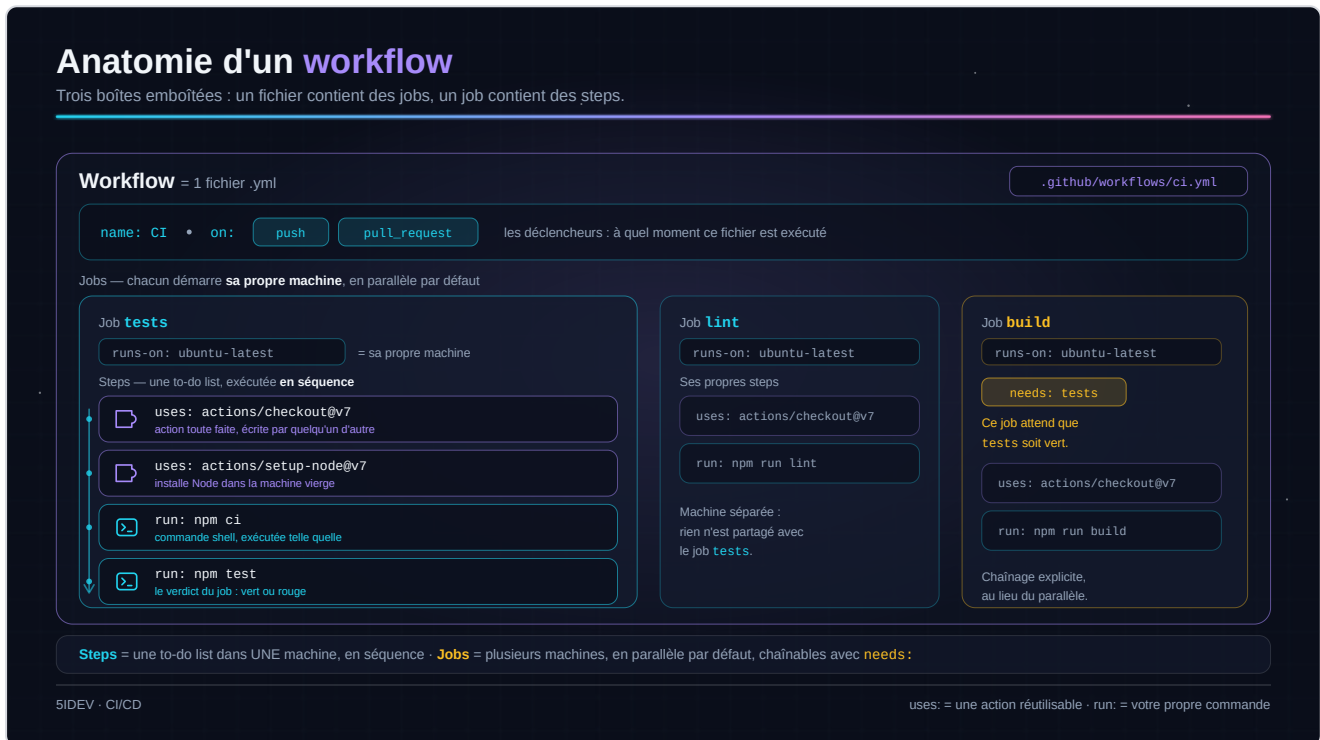
Quand vous poussez du code, GitHub regarde s'il y a des fichiers dans le dossier `.github/workflows/` de votre repo. Pour chaque workflow concerné par l'événement, il démarre une **machine virtuelle vierge** (par défaut Ubuntu), y clone votre code, exécute vos instructions dans l'ordre, note vert ou rouge, puis **détruit la machine**.

Trois conséquences très concrètes :

1. **Rien ne survit d'un run à l'autre.** Ni vos `node_modules`, ni votre base de données locale, ni votre fichier `.env`. Tout ce dont le job a besoin doit être décrit dans le YAML.

2. **C'est un ordinateur neutre.** S'il a réussi à installer et tester votre projet à partir de zéro, n'importe qui pourra le faire, y compris votre hébergeur. Si ça ne marche que sur votre machine, la CI vous le dira.
3. **Ça prend du temps et des ressources.** Un run typique dure 2 à 5 minutes. C'est gratuit et illimité pour les repos publics. Pour les repos privés, le plan gratuit offre 2 000 minutes par mois, avec des multiplicateurs pour Windows et macOS. Raison de plus pour rester sur Ubuntu et pour garder votre repo public.

Le vocabulaire : workflow, job, step



YAML

```

name: Node CI                                # le nom affiché dans l'onglet Actions
on: [push, pull_request]                     # QUAND ça tourne (les déclencheurs)

jobs:                                         # un workflow contient un ou plusieurs jobs
  tests:                                     # identifiant du job
    runs-on: ubuntu-latest                  # SUR QUOI ça tourne : une VM Ubuntu
    steps:                                  # la to-do list, exécutée dans l'ordre, dans cette
VM
      - uses: actions/checkout@v7 # une action toute faite : cloner le repo
      - uses: actions/setup-node@v7
        with:
          node-version: 24
      - run: npm ci                      # une commande shell, comme dans votre terminal
      - run: npm test

```

- Un **workflow** est un fichier YAML dans `.github/workflows/`. Un repo peut en avoir plusieurs (un par stack, un pour le déploiement...).

- Un **job** est une unité de travail qui tourne sur sa propre machine. Les jobs d'un même workflow tournent **en parallèle** par défaut.
- Un **step** est une étape d'un job. Les steps tournent **en séquence** dans la même machine et partagent son disque : ce que `npm ci` installe, `npm test` le trouve.
- Un step est soit `run:` (une commande shell), soit `uses:` (une **action**, c'est-à-dire un step tout fait publié par quelqu'un : cloner le repo, installer PHP, uploader un fichier...).

ASTUCE

Traduction honnête : les **steps**, c'est la to-do list dans **une** machine. Les **jobs**, ce sont **plusieurs** machines qui bossent en même temps. On parallélise ce qui est indépendant (lint et tests), on séquence ce qui dépend d'autre chose (déployer après les tests) avec `needs:` .

Quand est-ce que ça tourne ? Les déclencheurs

La clé `on:` décide des événements qui lancent le workflow. Les plus utiles :

Déclencheur	Se lance quand...	Pourquoi vous en voulez
<code>pull_request</code>	Une PR est ouverte ou reçoit un nouveau commit	Vérifier le code avant le merge. C'est le cœur de la CI
<code>push sur main</code>	Un merge (ou un push direct) arrive sur <code>main</code>	Vérifier <code>main</code> et déclencher le déploiement
<code>workflow_dispatch</code>	Vous cliquez sur « Run workflow » dans l'onglet Actions	Relancer à la main, tester le workflow sans pousser
<code>schedule</code>	Selon une expression cron	Vérifier chaque nuit que les dépendances n'ont pas cassé

`on: [push, pull_request]` est la forme courte : elle réagit à tous les push sur toutes les branches. On peut aussi détailler chaque déclencheur avec ses options, pour ne réagir qu'à certaines branches (`branches:`) ou qu'aux changements dans certains dossiers (`paths:`). Ci-dessous, le workflow Laravel du repo `cicd26` ne tourne que si le dossier `laravel/` ou le workflow lui-même change, ce qui évite de faire tourner les tests Laravel quand on ne touche qu'au front :

YAML

```
on:
  push:
    branches: [main]
    paths: ["laravel/**", ".github/workflows/laravel.yml"]
  pull_request:
    paths: ["laravel/**", ".github/workflows/laravel.yml"]
```

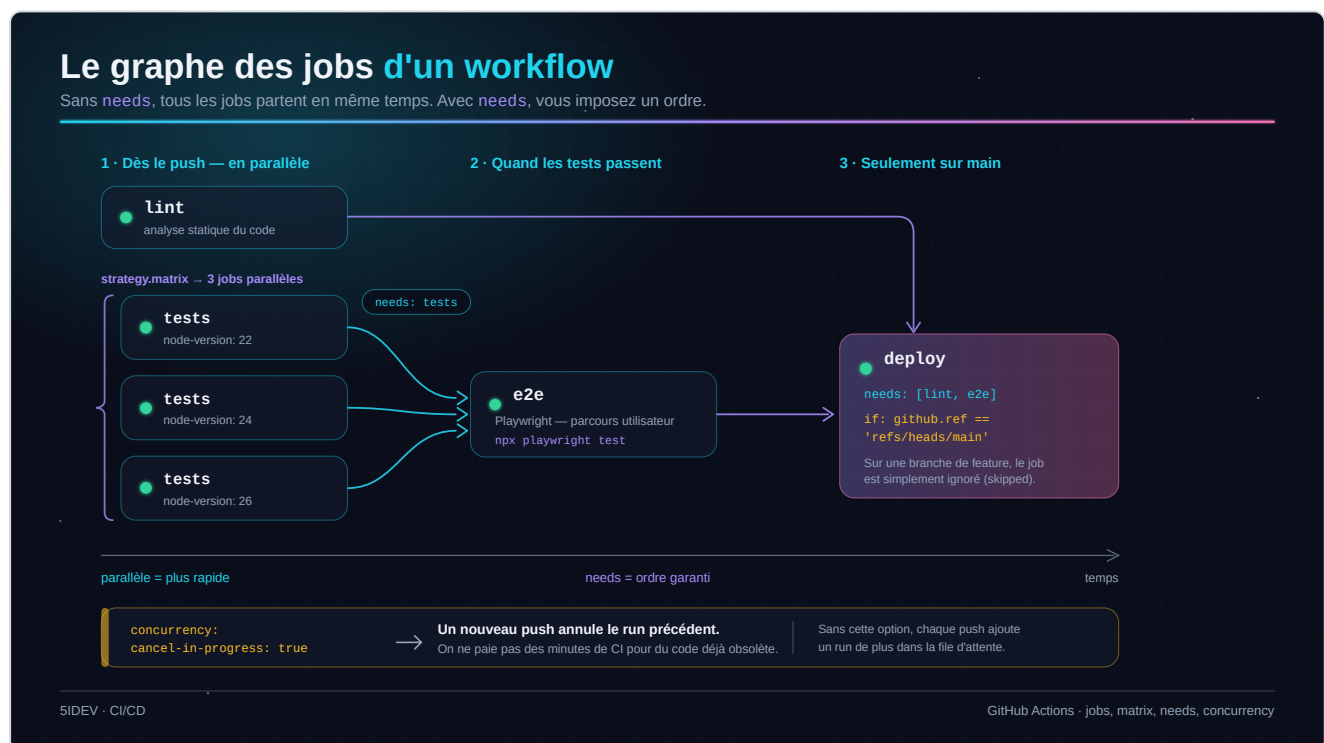
ATTENTION

Ne mettez pas `on: push` tout nu. Chaque push sur chaque branche lancerait un run, puis la PR en lancerait un deuxième. Ciblez `push` sur `main` et `pull_request` pour le reste : deux runs utiles, zéro run inutile.

Sur quelle machine ? Les runners

`runs-on` : choisit la machine. `ubuntu-latest` est le bon choix par défaut : le moins cher, le plus rapide à démarrer, et c'est un Linux comme votre serveur de production. `windows-latest` et `macos-latest` existent, mais ils coûtent 2 et 10 fois plus de minutes et ne vous apportent rien pour une application web. Ces machines viennent avec beaucoup d'outils préinstallés (Git, Docker, Node, Python, PHP dans plusieurs versions), mais vous devez toujours **choisir explicitement** la version avec une action `setup-*` : c'est la garantie que la CI teste la même version que votre prod.

Parallèle, séquence, matrices



Trois mécanismes organisent vos jobs :

needs : impose un ordre. Le job qui porte `needs: tests` attend que `tests` soit vert avant de démarrer, et n'est pas lancé du tout si `tests` est rouge. On l'utilise pour ne pas payer un test E2E lent ou un déploiement quand les unitaires échouent déjà.

YAML

```
jobs:
  tests:
    runs-on: ubuntu-latest
    steps: [...]
  e2e:
    needs: tests          # attend le vert de tests
    runs-on: ubuntu-latest
    steps: [...]
```

strategy.matrix multiplie un job. Vous écrivez le job une fois, GitHub le lance une fois par combinaison, en parallèle. C'est la manière standard de tester plusieurs versions d'un langage, ou plusieurs bases de données.

YAML

```
tests:
  runs-on: ubuntu-latest
  strategy:
    fail-fast: false      # si Node 22 échoue, on veut quand même le verdict de 24 et
26
    matrix:
      node: [22, 24, 26]
  steps:
    - uses: actions/setup-node@v7
      with:
        node-version: ${ matrix.node }
```

Pourquoi tester trois versions de Node ? Parce que votre hébergeur n'a peut-être pas la même que vous. En septembre 2026, Node 24 est la LTS active, Node 22 est en maintenance et Node 26 devient LTS en octobre. Tester les trois coûte trois fois plus de minutes mais vous évite la surprise au déploiement.

concurrency annule les runs devenus inutiles. Si vous poussez trois commits en cinq minutes sur la même PR, seul le dernier compte. Sans cette clé, les trois runs tournent jusqu'au bout.

YAML

```
concurrency:
  group: ci-${ github.ref }
  cancel-in-progress: true
```

Secrets et variables d'environnement

Votre code a besoin de valeurs qui ne doivent pas être dans le repo : mot de passe de base de données, clé d'API, `APP_KEY`. La règle du cours est claire : **jamais de .env commité**. Dans la CI, trois cas :

1. **Les valeurs de test sans importance** (`DB_PASSWORD: root` pour un MySQL jetable qui vit trois minutes) se mettent en clair dans le YAML, via `env: .`

2. **Les vraies valeurs sensibles** (un token de déploiement, une clé d'API tierce) se stockent dans *Settings* → *Secrets and variables* → *Actions* de votre repo, et se lisent avec `${{ secrets.NOM }}`. GitHub les masque dans les logs.
3. **Ce que votre framework attend dans un `.env`** se génère à la volée : `cp .env.example .env && php artisan key:generate` pour Laravel, par exemple. Votre `.env.example` doit donc rester à jour et complet : c'est lui que la CI lit.

```

env:                                     # visible par tous les steps du job
  DB_CONNECTION: mysql
  DB_PASSWORD: root                     # valeur jetable : en clair, pas grave
steps:
  - run: ./deploy.sh
    env:
      DEPLOY_TOKEN: ${ secrets.DEPLOY_TOKEN } # vraie valeur : un secret

```

YAML

Cache et artifacts

Le cache évite de retélécharger vos dépendances à chaque run. Les actions `setup-node`, `setup-python` le font pour vous avec un simple `cache: npm` ou `cache: pip` ; pour Composer on utilise `actions/cache@v6` avec une clé basée sur le hash de `composer.lock`. Quand le fichier de lock change, le cache est invalidé. C'est la différence entre 40 secondes et 8 secondes d'installation.

Les artifacts sont des fichiers produits par un job que vous voulez garder après la destruction de la machine : un rapport de tests, une couverture de code, les captures d'écran d'un test E2E qui a échoué. `actions/upload-artifact@v7` les rend téléchargeables depuis la page du run pendant quelques jours.

```

- uses: actions/upload-artifact@v7
  if: ${ !cancelled() } # même si les tests ont échoué : c'est justement là
    qu'on veut le rapport
  with:
    name: playwright-report
    path: playwright-report/
    retention-days: 14

```

YAML

4. Trois pipelines commentés ligne par ligne

Les trois workflows ci-dessous sont ceux du repo `cicd26`. Ils vivent dans un seul repo, d'où le `defaults.run.working-directory` qui fait tourner les commandes dans le bon sous-dossier ; dans votre projet, vous supprimerez cette clé et les filtres `paths`.

Chacun suit le même schéma : un job de tests rapide sur SQLite, un job de tests sur MySQL qui n'est lancé que si le premier passe, et un job de style en parallèle.

Pourquoi deux bases de données ?



SQLite en mémoire est parfait pour la CI : zéro installation, base neuve à chaque test, une suite complète en une seconde. Mais votre production tourne sur MySQL ou PostgreSQL, et les moteurs SQL ne sont pas interchangeables : une contrainte, un type de colonne, une fonction de date peuvent se comporter différemment. Tester **aussi** sur le vrai moteur attrape ces différences.

GitHub Actions le rend facile avec `services:` : vous déclarez une image Docker, GitHub la démarre à côté de votre job, dans la même VM, joignable sur `127.0.0.1`. Le `--health-cmd` est essentiel : MySQL met plusieurs secondes à démarrer, et sans lui votre premier test partirait avant que la base ne réponde.

YAML

```
services:
  mysql:
    image: mysql:8.4
    env:
      MYSQL_ROOT_PASSWORD: root
      MYSQL_DATABASE: chat
    ports: ["3306:3306"]
    options: >-
      --health-cmd="mysqladmin ping -h 127.0.0.1 -proot"
      --health-interval=10s
      --health-timeout=5s
      --health-retries=5
```

NOTE

Pour PostgreSQL, remplacez par `image: postgres:17`, les variables `POSTGRES_PASSWORD` et `POSTGRES_DB`, le port `5432` et `--health-cmd="pg_isready"`. Le principe est identique.

Laravel + Pest

YAML

```

name: Laravel CI

on:
  push:
    branches: [main]
  pull_request:

concurrency:
  group: laravel-{{ github.ref }}
  cancel-in-progress: true

jobs:
  tests:
    name: Tests (SQLite)
    runs-on: ubuntu-latest
    steps:
      - uses: actions/checkout@v7

      - name: Installer PHP
        uses: shivammathur/setup-php@v2          # LA référence pour PHP dans Actions
        with:
          php-version: "8.4"
          extensions: pdo_sqlite
          coverage: none                        # plus rapide sans Xdebug

      - name: Cache Composer
        uses: actions/cache@v6
        with:
          path: ~/.composer/cache
          key: composer-{{ hashFiles('composer.lock') }}

      - name: Installer les dépendances
        run: composer install --no-interaction --prefer-dist --no-progress

      - name: Préparer l'environnement
        run: |
          cp .env.example .env
          php artisan key:generate              # sans APP_KEY, Laravel refuse de
démarrer

      - name: Lancer les tests
        run: php artisan test                  # phpunit.xml force sqlite + :memory:

```

Ce qu'il faut voir :

- **shivammathur/setup-php@v2** est une action communautaire mais c'est le standard de fait : elle installe la version de PHP demandée et les extensions listées en quelques secondes.

- `cp .env.example .env puis key:generate` : la VM n'a pas votre `.env` . On le fabrique à partir de l'exemple. Si votre `.env.example` est incomplet, c'est ici que ça casse, et c'est une bonne chose de le découvrir.
- `php artisan test` lit `phpunit.xml`, qui définit `DB_CONNECTION=sqlite` et `DB_DATABASE=:memory:` pour les tests. Aucune base à créer.

Le job MySQL ajoute le bloc `services:` vu plus haut, `needs: tests`, et surcharge les variables :

```
tests-mysql:
  name: Tests (MySQL)
  runs-on: ubuntu-latest
  needs: tests
  services:
    mysql: { ... }
  env:
    DB_CONNECTION: mysql
    DB_HOST: 127.0.0.1
    DB_PORT: 3306
    DB_DATABASE: chat
    DB_USERNAME: root
    DB_PASSWORD: root
  steps:
    - uses: actions/checkout@v7
    - uses: shivammathur/setup-php@v2
      with: { php-version: "8.4", extensions: pdo_mysql, coverage: none }
    - run: composer install --no-interaction --prefer-dist --no-progress
    - run: cp .env.example .env && php artisan key:generate
    - run: php artisan test
```

*# le bloc complet est plus haut
gagnent sur phpunit.xml (qui n'a pas force="true")*

YAML

Et le job de style, indépendant, qui tourne en parallèle :

```
lint:
  name: Style (Pint)
  runs-on: ubuntu-latest
  steps:
    - uses: actions/checkout@v7
    - uses: shivammathur/setup-php@v2
      with: { php-version: "8.4", coverage: none }
    - run: composer install --no-interaction --prefer-dist --no-progress
    - run: vendor/bin/pint --test
```

--test : vérifie sans modifier, rouge si mal formaté

YAML

Pièges Laravel classiques en CI : `APP_KEY` manquante (oubli de `key:generate`), extension PHP absente (`pdo_mysql` non listée dans `extensions:`), test qui dépend d'un `php artisan migrate` que personne n'a lancé (utilisez `RefreshDatabase`, qui migre pour vous).

Django + pytest

YAML

```

name: Django CI

on:
  push:
    branches: [main]
  pull_request:

concurrency:
  group: django-
  github.refcancel-in-progress:truejobs:tests-sqlite:name:TestsSQLite(Python{{
github.ref }}
  cancel-in-progress: true

jobs:
  tests-sqlite:
    name: Tests SQLite (Python
github.refcancel-in-progress:truejobs:tests-sqlite:name:TestsSQLite(Python{{
matrix.python }})
    runs-on: ubuntu-latest
    strategy:
      matrix:
        python: ["3.12", "3.13", "3.14"]      # 3 jobs en parallèle
    steps:
      - uses: actions/checkout@v7

      - uses: actions/setup-python@v7
        with:
          python-version: ${{ matrix.python }}
          cache: pip      # cache basé sur requirements.txt

      - name: Dépendances système pour mysqlclient
        run: sudo apt-get update && sudo apt-get install -y default-libmysqlclient-dev
pkg-config

      - run: pip install -r requirements.txt
      - run: pytest -q

```

Ce qu'il faut voir :

- **La matrice Python** : le même job tourne trois fois, en parallèle. Le nom du job affiche la version, ce qui rend la liste des checks lisible dans la PR.
- `sudo apt-get install ...` : vous avez le droit d'installer des paquets système sur la VM, vous êtes root. Ici, `mysqlclient` a besoin des en-têtes MySQL pour se compiler. C'est un exemple typique de dépendance qui existe sur votre machine et pas sur une VM vierge.
- `pytest -q` lit `pytest.ini`, qui pointe vers `chatproject.settings`. La base de test SQLite est créée et détruite par `pytest-django`.

Le job MySQL reprend le `services:` et passe la configuration par variables d'environnement. Côté Django, `settings.py` bascule sur MySQL si `DB_ENGINE=mysql` est défini ; c'est une pratique à adopter dès maintenant, elle vous servira aussi chez votre hébergeur :

YAML

```
tests-mysql:
  name: Tests MySQL
  runs-on: ubuntu-latest
  needs: tests-sqlite
  services:
    mysql: { ... }
  env:
    DB_ENGINE: mysql
    DB_HOST: 127.0.0.1
    DB_PORT: 3306
    DB_NAME: chat
    DB_USER: root
    DB_PASSWORD: root
  steps:
    - uses: actions/checkout@v7
    - uses: actions/setup-python@v7
      with: { python-version: "3.13", cache: pip }
    - run: sudo apt-get update && sudo apt-get install -y default-libmysqlclient-dev
  pkg-config
    - run: pip install -r requirements.txt
    - run: pytest -q

lint:
  name: Style (ruff)
  runs-on: ubuntu-latest
  steps:
    - uses: actions/checkout@v7
    - uses: actions/setup-python@v7
      with: { python-version: "3.13" }
    - run: pip install ruff
    - run: ruff check .
```

Pièges Django classiques : `DJANGO_SETTINGS_MODULE` introuvable (pas de `pytest.ini`), `ALLOWED_HOSTS` ou `SECRET_KEY` lus depuis un `.env` absent (donnez des valeurs par défaut sûres pour les tests), migrations non committées (`makemigrations` oublié : la CI le verra tout de suite).

Node/Express + Vitest, puis Playwright

YAML

```

name: Node CI

on:
  push:
    branches: [main]
  pull_request:
  workflow_dispatch: # bouton « Run workflow »

concurrency:
  group: node-github.refcancel-in-progress:truejobs:unit:name:Vitest(Node{{ github.ref
}}
  cancel-in-progress: true

jobs:
  unit:
    name: Vitest (Node github.refcancel-in-progress:truejobs:unit:name:Vitest(Node{{
matrix.node }})
    runs-on: ubuntu-latest
    strategy:
      matrix:
        node: [22, 24, 26]
    steps:
      - uses: actions/checkout@v7
      - uses: actions/setup-node@v7
        with:
          node-version: ${{ matrix.node }}
          cache: npm
      - run: npm ci # installe EXACTEMENT package-
lock.json
      - run: npm test

```

Ce qu'il faut voir :

- **npm ci et pas npm install** : `ci` installe exactement ce que dit `package-lock.json` et échoue si le lock est absent ou incohérent. C'est ce que vous voulez : la CI teste les mêmes versions que vous, au patch près. Committez toujours votre lockfile.
- **cache: npm** : une ligne, et `setup-node` gère le cache pour vous.

Le job E2E est la partie la plus intéressante. Playwright ouvre un vrai navigateur Chromium, charge votre page, tape un message et vérifie qu'il apparaît. Le navigateur se fiche de votre stack serveur : ce job fonctionne à l'identique devant un backend Laravel ou Django. Il ne démarre que si les unitaires passent, et il publie son rapport HTML en artifact, échec ou pas.

YAML

```

e2e:
  name: Playwright (E2E)
  runs-on: ubuntu-latest
  needs: unit                                # les E2E sont lents : seulement si
les unitaires passent
  timeout-minutes: 15                        # un navigateur qui bloque ne doit pas
manger vos minutes
  steps:
    - uses: actions/checkout@v7
    - uses: actions/setup-node@v7
      with: { node-version: 24, cache: npm }
    - run: npm ci
    - run: npx playwright install --with-deps chromium # navigateur + librairies
système
    - run: npx playwright test
    - uses: actions/upload-artifact@v7
      if: ${{ !cancelled() }}
      with:
        name: playwright-report
        path: playwright-report/
        retention-days: 14

```

Le fichier `playwright.config.js` demande à Playwright de démarrer votre serveur lui-même et d'attendre que `/health` réponde avant de lancer le premier test. Vous n'avez donc rien à orchestrer dans le YAML :

JAVASCRIPT

```

export default defineConfig({
  testDir: "./e2e",
  use: { baseURL: "http://localhost:3000" },
  webServer: {
    command: "npm start",
    url: "http://localhost:3000/health",
    reuseExistingServer: !process.env.CI,
  },
});

```

Et le test lui-même tient en cinq lignes lisibles par n'importe qui :

JAVASCRIPT

```

test("un utilisateur envoie un message et le voit apparaître", async ({ page }) => {
  await page.goto("/");
  await page.fill("#body", "Bonjour depuis Playwright");
  await page.click("button[type=submit]");
  await expect(page.locator("#messages li").last()).toContainText("Bonjour depuis
Playwright");
});

```

Pièges Node classiques : lockfile absent ou non committé (`npm ci` refuse), test qui dépend d'un port déjà pris, E2E lancés sans `--with-deps` (Chromium a besoin de librairies système que la VM n'a pas).

ASTUCE

Un seul test E2E sur votre parcours vital (se connecter, envoyer un message, le voir arriver) vaut plus que dix tests unitaires supplémentaires. Il vérifie que tout est branché : routes, base, JavaScript côté client, WebSocket. C'est aussi une superbe démo pour votre rapport.

5. Le CD : brancher la production



Vous avez maintenant une CI qui dit vert ou rouge sur chaque PR. Reste à faire en sorte que **le vert déploie** et que **le rouge bloque**.

Votre hébergeur déploie déjà à chaque push

Bonne nouvelle : vous n'avez presque rien à écrire. Tous les hébergeurs que vous avez comparés au premier cours proposent le *push-to-deploy* :

- Les PaaS managés** (Vercel, Netlify, Render, Railway, Fly.io, Heroku...) se connectent à votre repo GitHub en trois clics : choisissez le repo et la branche `main`, l'hébergeur détecte la stack, construit et déploie à chaque push. La plupart offrent aussi des **preview deployments** : chaque PR reçoit une URL temporaire, parfaite pour montrer une fonctionnalité à votre binôme avant le merge.

- **Un VPS** (Hetzner, OVH, DigitalOcean...) vous donne une machine nue. Pour retrouver le même confort, installez un panneau de déploiement auto-hébergé comme **Coolify** ou **Dokploy** : une interface web, une intégration GitHub, un bouton « déployer à chaque push », la gestion des variables d'environnement, des bases de données et des certificats HTTPS. Vous gardez le contrôle et le prix fixe du VPS, avec l'expérience d'un PaaS.

Dans les deux cas, le mécanisme est le même : l'hébergeur écoute les événements de votre repo via un *webhook* GitHub. Push sur `main` → il clone, construit, redémarre.

Faire attendre la CI

Il y a un piège. Par défaut, l'hébergeur déploie **dès le push**, sans attendre votre workflow. Si un push cassé arrive sur `main`, il est en production avant que la CI ait fini de dire rouge. Deux façons de l'éviter :

1. **Ne jamais pousser directement sur `main`**. Tout passe par une PR, et la PR ne peut être mergée que si la CI est verte. C'est la protection de branche, ci-dessous. Une fois en place, tout ce qui arrive sur `main` a déjà été validé et l'hébergeur peut déployer immédiatement.
2. **Demander à l'hébergeur d'attendre**. Beaucoup le proposent en une case à cocher (« Wait for CI to pass » chez Render et Railway, par exemple). Sinon, désactivez le déploiement automatique et déclenchez-le depuis un job `deploy` de votre workflow, avec `needs:` sur vos tests, en appelant l'URL de *deploy hook* fournie par l'hébergeur :

```

deploy:
  needs: [tests, tests-mysql, lint]
  if: github.ref == 'refs/heads/main'           # jamais depuis une PR
  runs-on: ubuntu-latest
  steps:
    - run: curl -fsS -X POST "${{ secrets.DEPLOY_HOOK_URL }}"

```

YAML

L'URL du hook est un secret : quiconque la connaît peut déclencher un déploiement.

La protection de branche : rendre le rouge bloquant

Sans cette étape, votre CI est décorative : elle affiche une croix rouge et n'importe qui peut cliquer sur « Merge » quand même. Dans votre repo : *Settings* → *Branches* (ou *Rules* → *Rulesets* sur les repos récents) → ajoutez une règle sur `main` avec :

- **Require a pull request before merging** : plus de push direct sur `main` ;
- **Require status checks to pass** : cochez vos jobs (`Tests (SQLite)` , `Style (Pint)` ...). Le bouton « Merge » reste gris tant qu'ils ne sont pas verts ;
- **Require approvals : 1**, si vous voulez formaliser la review par le binôme, ce qui colle à la consigne d'au moins une PR revue par personne et par semaine.

Ces trois cases sont, à elles seules, la moitié de la valeur de ce chapitre. Cochez-les aujourd'hui.

3. **Dans le log**, cherchez la **première** ligne d'erreur, pas la dernière. La dernière ligne dit toujours `Process completed with exit code 1`, ce qui ne vous apprend rien. Le vrai message est plus haut : un nom de test, une assertion, une exception.
4. **Reproduisez en local** avec exactement la même commande que le YAML. Si ça passe chez vous et pas en CI, la différence est dans l'environnement : une version, une variable d'environnement, un fichier non committé, une dépendance système.
5. **Re-run jobs** uniquement si l'échec est aléatoire : un téléchargement qui a échoué, un timeout réseau. Un test qui échoue une fois sur trois est un bug, dans le test ou dans le code. Ne le relancez pas jusqu'à ce qu'il passe : corrigez-le.

Quelques causes fréquentes, dans l'ordre où vous les rencontrerez :

Symptôme	Cause probable
Échec dès <code>npm ci</code> ou <code>composer install</code>	Lockfile absent, non committé ou désynchronisé
<code>APP_KEY</code> , <code>SECRET_KEY</code> , variable introuvable	<code>.env.example</code> incomplet, ou une variable définie seulement dans votre <code>.env</code> local
<code>Connection refused</code> sur la base	Service sans health check, ou mauvais <code>DB_HOST</code> (c'est <code>127.0.0.1</code> , pas <code>localhost</code> ni <code>mysql</code>)
Tests verts en local, rouges en CI	Ordre des tests, base non remise à zéro, fuseau horaire, fichier généré localement
Un job qui n'apparaît pas du tout	Il a un <code>needs:</code> sur un job rouge, ou un filtre <code>paths:</code> qui l'exclut

ATTENTION

N'éditez jamais le workflow « au hasard » pour faire passer la CI. Si vous ne comprenez pas l'erreur, vous ne comprenez pas non plus ce que vous êtes en train de désactiver. Lisez le log, reproduisez, corrigez la cause.

7. À vous : TP et checklist Projet 1

TP en classe (30 minutes)

Sur votre projet, en binôme :

1. **Créez** `.github/workflows/ci.yml` en partant du modèle de votre stack. Retirez `working-directory` et les filtres `paths`. Poussez sur une branche, ouvrez une PR, regardez le premier run. Il sera probablement rouge : c'est normal, lisez le log, corrigez jusqu'au vert.

2. **Écrivez un test** sur une règle réelle de votre chat : un message vide est refusé, ou un non-membre reçoit 403. Faites-le passer.
3. **Cassez-le volontairement** : changez la règle dans le code, pas dans le test. Poussez. La PR passe au rouge et le bouton « Merge » se grise si la protection de branche est active. Lisez le log, trouvez la ligne utile, remettez le code en état, revoyez le vert.
4. **Activez la protection de branche** sur `main` avec vos checks obligatoires.
5. **Vérifiez le déploiement** : mergez, regardez votre hébergeur construire, ouvrez l'URL de production.

Checklist avant la revue d'avancement n°1

- ☐ Un workflow dans `.github/workflows/` qui tourne sur `pull_request` et sur `push` vers `main`
- ☐ Au moins un test d'intégration sur le cœur (envoi ou lecture de message, permissions), qui passe en CI
- ☐ Un job de style (Pint, ruff ou ESLint) en parallèle des tests
- ☐ La base de données de test décrite dans le YAML (SQLite en mémoire ou service container)
- ☐ Aucune valeur sensible dans le repo : secrets GitHub et `.env.example` à jour
- ☐ Protection de branche sur `main` : PR obligatoire, checks obligatoires
- ☐ Déploiement automatique sur `main` chez votre hébergeur, vérifié au moins une fois
- ☐ Badge de statut dans le README

Le rapport de projet contient une section « CI/CD et déploiement » : cette checklist en est le plan. Une capture d'un run vert, une capture de la règle de protection de branche, trois phrases sur ce que vos tests couvrent, et le jury sait que vous avez compris.

En résumé

- La CI attrape les bugs **dans la PR**, là où ils coûtent quelques minutes au lieu d'une soirée.
- Un workflow GitHub Actions est un fichier YAML : `on:` dit quand, `jobs:` tournent en parallèle sur des VM vierges, `steps:` s'exécutent en séquence dans chacune. `needs:` ordonne, `matrix` multiplie, `concurrency` annule l'inutile.
- Rien ne survit d'un run à l'autre : dépendances, `.env`, base de données, tout doit être décrit dans le YAML. Ce qui casse en CI est ce qui aurait cassé chez l'hébergeur.
- Testez le cœur de votre application, pas ses getters. Un test d'intégration par règle métier vitale, un E2E sur le parcours principal.
- SQLite pour la vitesse, un service container MySQL ou PostgreSQL pour la fidélité. Le health check n'est pas optionnel.
- Votre hébergeur déploie déjà à chaque push. Votre travail, c'est de garantir que rien n'arrive sur `main` sans être vert : protection de branche, checks obligatoires.

- Un run rouge se lit de haut en bas, première erreur d'abord, et se reproduit en local avant toute modification du workflow.

Prochain point de théorie : le lint et l'analyse statique, ou comment faire relire tout votre code par une machine qui ne se fatigue jamais.

Pour aller plus loin

- Documentation GitHub Actions : <https://docs.github.com/actions>
- Syntaxe complète d'un workflow : <https://docs.github.com/actions/reference/workflow-syntax-for-github-actions>
- Service containers : <https://docs.github.com/actions/use-cases-and-examples/using-containerized-services>
- Playwright en CI : <https://playwright.dev/docs/ci-intro>
- Pest : <https://pestphp.com> · pytest-django : <https://pytest-django.readthedocs.io> · Vitest : <https://vitest.dev>
- Coolify : <https://coolify.io> · Dokploy : <https://dokploy.com>
- Repo d'exemple du cours : <https://github.com/opmvpc/cicd26>

1.4 Lint, formatage et analyse statique : faire relire votre code par une machine

Point de théorie · à brancher dans votre pipeline CI dès que les tests y tournent

Slides du cours

-  [deck4-lint.pdf](#) PDF — les 14 slides du point de théorie

Dans le chapitre CI/CD, vous avez appris à faire vérifier votre code par des tests. Ce chapitre présente une autre famille de vérifications, complémentaire et presque gratuite : des outils qui **lisent votre code sans l'exécuter** et y trouvent des erreurs, des incohérences et des fautes de style, en quelques secondes, sur l'ensemble du projet. Y compris sur les parties que vous n'avez pas testées.

À la fin de ce chapitre, vous serez capables de :

1. **distinguer** formatage, lint et analyse statique, et dire quel type d'erreur chacun attrape ;
2. **expliquer** comment un analyseur statique trouve un bug sans exécuter le programme, et **lire** le message qu'il produit ;
3. **installer et configurer** les outils de référence de votre stack (Laravel, Django ou Node/TypeScript) ;

4. les **brancher** dans votre éditeur et dans votre pipeline GitHub Actions ;
5. les **adopter progressivement** sur un projet existant sans vous noyer sous les erreurs.

1. Trois familles d'outils, trois questions

Deux scènes de la vie d'un binôme

Mardi. Votre binôme ouvre une PR de vingt lignes. GitHub en affiche trois cents : son éditeur a réindenté tout le fichier et remplacé les guillemets simples par des doubles. Vous passez dix minutes à chercher les vingt lignes qui comptent, vous en ratez une, vous approuvez.

Jeudi. Un utilisateur tape une URL de conversation avec un identifiant qui n'existe pas. Erreur 500. Le code qui plante n'était pas testé, personne n'a pensé au cas. Il aurait suffi qu'un outil signale, au moment où la ligne a été écrite, que la variable pouvait être `null`.

Ces deux problèmes ont la même solution : des outils qui lisent le code à votre place. Et vous en utilisez déjà un sans le savoir : **le soulignement rouge dans VS Code**, quand vous appelez une méthode qui n'existe pas, c'est de l'analyse statique. Ce chapitre explique ce qui se passe derrière et comment l'étendre à tout le projet et à toute l'équipe.

Trois questions, trois outils

Quand on dit « le lint », on mélange souvent trois choses qui n'ont pas le même rôle. Les séparer permet de comprendre ce que chaque outil vous apporte et pourquoi on les cumule.

Trois familles d'outils

Chacune répond à une question différente. Elles ne se remplacent pas : elles s'empilent.

Formatage

« Est-ce écrit de manière uniforme ? »

EXEMPLES

- indentation
- guillemets
- virgule finale

ACTION

réécrit le code, sans changer son comportement

OUTILS

Pint ruff format Prettier

Lint

« Y a-t-il des motifs suspects ? »

EXEMPLES

- variable inutilisée
- import inutile
- == au lieu de ===

ACTION

signale, corrige parfois (option --fix)

OUTILS

Pint ruff check ESLint

Analyse statique

« Les types collent-ils ? »

EXEMPLES

- méthode sur un objet peut-être null
- mauvais type d'argument
- propriété inexistante

ACTION

prouve une erreur avant toute exécution

OUTILS

PHPStan mypy tsc

LE MÊME CODE, TROIS VERDICTS

```
$conversation = Conversation::find($id); return $conversation->name;
```

✓ **formateur** — rien à dire

✓ **linter** — rien à dire

ⓘ **analyse statique**
Cannot access property \$name on Conversation|null

SIDEV · Qualité du code

Trois filets successifs : forme, motifs, types.

Famille	La question qu'elle pose	Exemple de ce qu'elle attrape	Ce qu'elle fait
Formatage	« Ce code est-il écrit de manière uniforme ? »	Indentation à 2 ou 4 espaces, guillemets simples ou doubles, virgule finale, longueur de ligne	Réécrit le code automatiquement, sans changer son comportement
Lint	« Y a-t-il des motifs suspects ou interdits ? »	Variable jamais utilisée, <code>import</code> inutile, <code>=</code> au lieu de <code>===</code> , <code>console.log</code> oublié, fonction trop longue	Signale (et corrige parfois) des règles que l'équipe a choisies
Analyse statique	« Ce code est-il cohérent ? Les types collent-ils ? »	Méthode appelée sur un objet qui peut être <code>null</code> , argument de type texte passé à une fonction qui attend un entier, propriété qui n'existe pas	Prouve une erreur avant toute exécution, en raisonnant sur les types

Un exemple concret dans votre chat. Le contrôleur qui affiche une conversation :

PHP

```
public function show(int $id)
{
    $conversation = Conversation::find($id); // peut renvoyer null

    return view('chat.show', ['title' => $conversation->name]);
}
```

- Le **formateur** ne dira rien : ce code est proprement écrit.
- Le **linter** ne dira rien non plus : aucun motif suspect.
- L'**analyseur statique** dira : « *Cannot access property \$name on Conversation|null* ». Il a lu la signature de `find()`, a vu qu'elle peut renvoyer `null`, et a constaté que vous ne vérifiez pas ce cas. C'est exactement l'erreur 500 de jeudi.

Le même bug existe dans chaque stack, et chaque analyseur le formule à sa manière :

Stack	Le code qui plante	Ce que dit l'analyseur
Laravel (PHPStan)	<code>Conversation::find(\$id)->name</code>	<code>'Cannot access property \$name on Conversation'</code>
Django (mypy)	<code>Conversation.objects.filter(pk=id).first().name</code>	<code>'Item "None" of "Conversation'</code>
TypeScript (tsc)	<code>conversations.find((c) => c.id === id).name</code>	<code>Object is possibly 'undefined'</code>

Trois formulations, une seule idée : *cette valeur peut être vide, et vous faites comme si elle ne l'était jamais*. La correction est toujours la même : traiter le cas vide (`findOrFail`, `get_object_or_404`, un `if`), et l'outil se tait.

NOTE

Les trois familles se recouvrent un peu : ruff fait du lint **et** du formatage, ESLint a des règles de style, PHPStan a des règles de lint. Ce qui compte, c'est de couvrir les trois questions, pas de savoir dans quelle case ranger chaque outil.

2. Pourquoi c'est important en équipe

Le formateur met fin aux débats inutiles

Sans formateur, chaque membre du groupe code avec les réglages de son éditeur. Résultat : une PR qui touche trois lignes en affiche trois cents modifiées parce que l'éditeur de l'un a réindenté le fichier de l'autre. La review devient impossible, on ne voit plus le changement réel.

Avec un formateur, **la question du style disparaît**. Vous ne discutez plus de l'emplacement des accolades : l'outil décide, tout le monde applique, les diffs ne montrent que ce qui change vraiment. C'est la première chose à installer sur un projet en groupe, avant même le premier commit.

Le linter capture l'expérience des autres

Chaque règle de lint est un bug que quelqu'un a déjà eu. `no-unused-vars` existe parce que des milliers de développeurs ont laissé traîner une variable qu'ils croyaient utiliser. `eqeqeq` existe parce que `'0' === false` vaut `true` en JavaScript. Activer un jeu de règles recommandé, c'est bénéficier de cette expérience sans avoir à payer pour l'acquérir.

L'analyse statique teste ce que vous n'avez pas testé

Vos tests couvrent les chemins que vous avez pensé à écrire. L'analyseur statique parcourt **tout le code**, y compris les branches d'erreur, les cas limites et les fichiers que personne n'a ouverts depuis trois semaines. Il ne remplace pas les tests : il ne sait pas si votre logique métier est correcte. Mais il garantit qu'elle est **cohérente**, et cette garantie vaut sur cent pour cent du projet.

ASTUCE

Les trois outils font gagner du temps à la **review**. Quand la machine a déjà vérifié le style, les motifs suspects et la cohérence des types, votre binôme peut consacrer sa relecture à ce qu'aucun outil ne voit : est-ce la bonne solution, est-ce lisible, est-ce ce que l'utilisateur voulait ?

3. Comment un analyseur statique trouve un bug sans exécuter le code



Le principe tient en trois étapes.

- Il lit le code comme le fait le langage lui-même**, et construit un arbre qui représente sa structure : ici une fonction, là un appel de méthode, là une condition. C'est l'*arbre syntaxique*, celui que l'interpréteur PHP ou Python construit avant d'exécuter quoi que ce soit.
- Il déduit le type de chaque expression.** `Conversation::find($id)` : il ouvre la définition de `find()` dans le framework, lit son type de retour (`?Conversation`, c'est-à-dire `Conversation` ou `null`), et retient que `$conversation` peut être `null`. Il propage ainsi les types dans tout le programme, en suivant les affectations, les conditions et les retours de fonction.
- Il vérifie que chaque usage est compatible avec le type déduit.** Accéder à `->name` sur quelque chose qui peut être `null` : incompatible, il le signale. Passer une chaîne à une fonction qui attend un entier : incompatible. Appeler une méthode qui n'existe pas dans la classe : incompatible.

Tout cela sans lancer le programme, sans base de données, sans navigateur. C'est pourquoi c'est rapide (quelques secondes sur un projet étudiant) et pourquoi ça couvre tout le code.

C'est aussi exactement ce que fait votre éditeur en continu : l'autocomplétion qui vous propose `->name` et pas `->nam`, le soulignement rouge sur une méthode inconnue, le survol qui affiche un type. VS Code embarque un analyseur statique par langage (Intelephense pour PHP, Pylance pour Python, le serveur TypeScript pour JS/TS). Les outils de ce chapitre sont les mêmes moteurs, ou leurs cousins, lancés sur tout le projet en ligne de commande, avec un verdict vert ou rouge que la CI peut lire.

Lire un message d'outil

Tous ces outils parlent la même langue : **un fichier, une ligne, un code de règle, un message**. Trois exemples réels :

```
chat/views.py:3:8: F401 [*] `json` imported but unused
src/app.js 14:9 error 'user' is assigned a value but never used no-unused-vars
app/Http/Controllers/MessageController.php:21 Cannot access property $name on
Conversation|null.
```

- Le **code de règle** (F401 , no-unused-vars) est une clé de recherche : collez-le dans la doc de l'outil et vous obtenez la page qui explique la règle, pourquoi elle existe et comment la satisfaire. Prenez l'habitude de lire cette page avant de désactiver quoi que ce soit.
- Le **[*]** de ruff, ou le mot *fixable* d'ESLint, signale une correction automatique disponible avec `--fix`.
- PHPStan n'affiche pas de code par défaut mais son message est une phrase complète ; l'option `--error-format=table` le rend plus lisible en local.

Un message que vous ne comprenez pas n'est jamais une raison de désactiver la règle : c'est une raison d'ouvrir sa doc, ou de demander à votre binôme.

Le rôle des annotations de types

Plus vous donnez d'information de types, plus l'analyseur voit loin. `function total($items)` ne lui dit rien ; `function total(array $items): float` lui permet de vérifier chaque appel. En PHP et en Python, les types sont optionnels : l'analyseur fait ce qu'il peut avec ce qu'il a, et vous encourage à en ajouter. En TypeScript, les types sont le cœur du langage : le compilateur `tsc` **est** l'analyseur statique.

Les niveaux de sévérité

Un analyseur qu'on lance pour la première fois sur un projet existant remonte des centaines d'erreurs. Pour ne pas décourager, les outils proposent des niveaux :

- **PHPStan** a des niveaux de 0 à 10. Le niveau 0 vérifie seulement que les classes et méthodes appelées existent ; le niveau 5 vérifie les types des arguments ; le niveau 9 traque le moindre `mixed`. On commence bas et on monte d'un cran quand le projet est propre.
- **mypy** distingue le mode par défaut, indulgent avec le code non annoté, et `--strict`, qui exige des types partout.
- **TypeScript** a `strict: true` dans `tsconfig.json`, qui active tout le jeu de vérifications strictes, dont `strictNullChecks`, le plus utile.

Les faux positifs existent : l'outil signale parfois un cas que vous savez impossible. Chaque outil permet de l'ignorer localement avec un commentaire explicite (`// @phpstan-ignore-next-line` , `# type: ignore` , `// @ts-expect-error`). La règle d'équipe : **on ignore une erreur avec un commentaire qui dit pourquoi**, jamais en silence, jamais en baissant le niveau global.

4. Les outils de référence par stack

Pour chaque stack, trois outils, une installation, une configuration minimale et la commande que vous lancerez en local et en CI. Le repo <https://github.com/opmvpc/cicd26> les intègre.

Stack	Formatage	Lint	Analyse statique
Laravel	Pint	Pint (règles du preset) et PHPStan	PHPStan avec Larastan
Django	ruff format	ruff check	mypy avec django-stubs, ou pyright
Node / TypeScript / Vue	Prettier	ESLint (avec typescript-eslint et eslint-plugin-vue)	TypeScript, <code>tsc --noEmit</code> ou <code>vue-tsc</code>

Laravel : Pint et PHPStan (Larastan)

Pint est le formateur officiel de Laravel, déjà présent dans tout nouveau projet. C'est un PHP-CS-Fixer préconfiguré : il applique le style de code du framework et, au passage, une partie de ce qu'un linter ferait (imports inutilisés supprimés, ordre des imports, syntaxe modernisée). C'est pourquoi il apparaît dans deux colonnes du tableau : en PHP, formatage et lint sont assurés par le même outil. Sa configuration tient dans `pint.json` à la racine, et la version par défaut suffit largement :

```
{
  "preset": "laravel"
}
```

JSON

```
vendor/bin/pint          # reformate tout le projet
vendor/bin/pint --test   # vérifie sans modifier : la commande de la CI
vendor/bin/pint --dirty  # ne touche qu'aux fichiers modifiés depuis le dernier
                        commit
```

BASH

PHPStan est l'analyseur statique de référence en PHP. **Larastan** est son extension pour Laravel : elle lui apprend les facades, Eloquent, les relations, les helpers, tout ce qui est magique dans le framework et que PHPStan seul ne comprendrait pas.

BASH

```
composer require --dev larastan/larastan
```

Fichier `phpstan.neon` à la racine :

```
includes:
  - vendor/larastan/larastan/extension.neon

parameters:
  level: 5
  paths:
    - app
    - routes
```

BASH

```
vendor/bin/phpstan analyse --memory-limit=1G
```

Le format `.neon` est un cousin du YAML propre à PHPStan. `--memory-limit=1G` lève la limite mémoire de PHP en ligne de commande, souvent fixée à 128 Mo : PHPStan charge tout le framework en mémoire pour le lire, et s'arrête net sans cette option.

On analyse le code de l'application, pas les tests : dans les tests Pest, `$this` est étendu dynamiquement et PHPStan ne le comprend pas sans plugin supplémentaire. Commencez par `app` et `routes`, c'est là que vivent les bugs qui atteignent la production.

Sur le contrôleur de l'exemple, PHPStan niveau 5 remonte l'accès à `→name` sur un objet potentiellement `null`. La correction est celle que le framework propose depuis toujours : `Conversation::findOrFail($id)`, qui renvoie une 404 propre au lieu d'une 500.

Django : ruff et mypy

ruff a remplacé, à lui seul, une demi-douzaine d'outils Python historiques (flake8, isort, black, pyupgrade...). Il fait le lint **et** le formatage, il est extrêmement rapide, et sa configuration vit dans `pyproject.toml` :

```
[tool.ruff]
line-length = 120
exclude = ["**/migrations", ".venv"]

[tool.ruff.lint]
select = ["E", "F", "I", "B", "UP", "DJ"] # erreurs, pyflakes, imports, bugbear,
modernisation, Django
```

BASH

```

pip install ruff
ruff format .          # formate
ruff check .           # lint : la commande de la CI
ruff check --fix .     # corrige ce qui peut l'être automatiquement

```

Les règles `DJ` sont propres à Django : elles signalent par exemple un modèle sans `__str__` ou un champ `null=True` sur un `CharField`.

mypy est l'analyseur de types de référence en Python. Comme PHPStan, il a besoin d'une extension pour comprendre la magie du framework : **django-stubs**.

BASH

```

pip install mypy django-stubs

```

Dans `pyproject.toml` :

```

[tool.mypy]
plugins = ["mypy_django_plugin.main"]
check_untyped_defs = true

[tool.django-stubs]
django_settings_module = "chatproject.settings"

```

BASH

```

mypy .

```

`check_untyped_defs = true` demande à mypy de vérifier aussi l'intérieur des fonctions sans annotations, au lieu de les ignorer : c'est ce qui rend l'outil utile sur un projet où vous n'avez pas encore tout typé. Pour qu'il voie vraiment loin, annotez vos fonctions : `def validate_body(body: object) → str | None: .` Sans annotation, il considère tout comme `Any` et ne peut rien prouver. **pyright** est une alternative plus rapide et plus stricte, intégrée à VS Code via l'extension Pylance ; les deux se configurent de manière similaire.

ATTENTION

Piège rencontré en préparant `cicd26` : `django-stubs` donne des types génériques aux champs (`ManyToManyField[Member, Conversation]`). Si vous recopiez cette annotation dans un modèle, ajoutez `from __future__ import annotations` en tête du fichier, sinon Python évalue le générique à l'exécution et plante au démarrage. Les erreurs d'analyse statique se corrigent dans le code, mais on relance toujours les tests après : un outil peut vous pousser vers une syntaxe que le runtime ne connaît pas.

Node, TypeScript et Vue : Prettier, ESLint, tsc

Deux parcours selon votre projet. **En JavaScript pur** (une API Express sans TypeScript, comme le dossier `node/` de `cicd26`) : Prettier et ESLint suffisent, l'analyse de types reste celle de l'éditeur.

En TypeScript ou en Vue avec TypeScript : on ajoute `tsc`, qui devient votre analyseur statique. Personne ne vous demande de migrer vers TypeScript pour ce chapitre.

Prettier formate JavaScript, TypeScript, Vue, CSS, JSON et Markdown. Il est volontairement peu configurable : c'est le principe, on arrête de discuter. Un fichier `.prettierrc` de deux lignes suffit :

JSON

```
{ "semi": true, "singleQuote": false }
```

BASH

```
npm install -D prettier
npx prettier --write .      # formate
npx prettier --check .      # vérifie : la commande de la CI
```

ESLint est le linter. Depuis la version 9, il se configure dans `eslint.config.js` (la *flat config*). Pour un projet TypeScript avec Vue :

BASH

```
npm install -D eslint @eslint/js typescript-eslint eslint-plugin-vue eslint-config-prettier
```

JAVASCRIPT

```
import js from "@eslint/js";
import prettier from "eslint-config-prettier";
import vue from "eslint-plugin-vue";
import tseslint from "typescript-eslint";

export default tseslint.config(
  js.configs.recommended,
  ...tseslint.configs.recommended,
  ...vue.configs["flat/recommended"],
  prettier, // désactive les règles de style qui
             // feraient doublon avec Prettier
  { ignores: ["dist/", "node_modules/"] },
);
```

`eslint-config-prettier` est là pour une raison précise : ESLint a historiquement des règles de style (point-virgule, guillemets) qui entreraient en conflit avec Prettier. Ce paquet les éteint, chacun reste dans son rôle. Pour un projet JavaScript sans Vue ni TypeScript, retirez les lignes `tseslint` et `vue`, et remplacez `tseslint.config()` par un simple tableau.

BASH

```
npx eslint .
```

TypeScript est l'analyseur statique. Si votre projet est en TypeScript, le compilateur fait le travail ; on le lance sans produire de fichiers :

```
npx tsc --noEmit          # projet TS
npx vue-tsc --noEmit      # projet Vue : vérifie aussi les <template>
```

BASH

`--noEmit` veut dire « vérifie mais n'écris aucun fichier `.js` » : c'est Vite qui compile votre projet, `tsc` ne sert ici qu'à juger.

Et dans `tsconfig.json`, une seule option compte vraiment :

```
{ "compilerOptions": { "strict": true } }
```

JSON

Si votre projet est en JavaScript pur, vous pouvez quand même profiter de l'analyse statique : ajoutez `// @ts-check` en tête d'un fichier et VS Code vérifie les types déduits de votre code et des bibliothèques. C'est une bonne porte d'entrée vers TypeScript.

5. Où brancher ces outils



Le même outil s'utilise à trois moments, de plus en plus tard, de plus en plus contraignant.

Dans l'éditeur, en tapant

C'est là que les outils rendent le plus de service : l'erreur apparaît soulignée pendant que vous écrivez, avant même de sauvegarder. Installez les extensions VS Code de votre stack (Prettier, ESLint, Pylance ou ruff, PHP Intelephense ou l'extension PHPStan) et activez le **formatage à la sauvegarde** dans vos réglages :

```

{
  "editor.formatOnSave": true,
  "[javascript][typescript][vue]": { "editor.defaultFormatter": "esbenp.prettier-vscode" },
  "[python]": { "editor.defaultFormatter": "charliermarsh.ruff" },
  "[php]": { "editor.defaultFormatter": "open-southerners.laravel-pint" }
}

```

Committez ce fichier dans `.vscode/settings.json`, et à côté un `.vscode/extensions.json` qui liste les extensions du projet : VS Code proposera de les installer à quiconque ouvre le dossier. Tout le groupe a la même expérience sans rien configurer, et un nouveau membre est opérationnel en deux clics.

```

{
  "recommendations": ["esbenp.prettier-vscode", "dbaeumer.vscode-eslint",
    "charliermarsh.ruff", "ms-python.vscode-pylance", "bmewburn.vscode-intelephense-client", "open-southerners.laravel-pint"]
}

```

Gardez seulement les extensions de votre stack.

Au commit, avec un hook

Un *hook* de pre-commit lance le formateur et le linter sur les fichiers modifiés juste avant que le commit ne soit créé. Si ça échoue, le commit est refusé. C'est optionnel, mais ça évite de pousser du code que la CI va refuser cinq minutes plus tard.

- **Node** : `husky` et `lint-staged` s'installent en deux commandes et ne lancent les outils que sur les fichiers concernés par le commit. La configuration tient dans `package.json` :

```

"lint-staged": {
  "*.{js,ts,vue}": ["prettier --write", "eslint --fix"]
}

```

- **Python** : le framework `pre-commit` fait la même chose à partir d'un fichier `.pre-commit-config.yaml`.

- **Laravel** : un hook shell qui appelle `vendor/bin/pint --dirty` suffit, ou le paquet `husky` si votre projet a déjà un `package.json`.

ATTENTION

Un hook qui prend trente secondes sera contourné par toute l'équipe avec `--no-verify`. Ne mettez dans le hook que ce qui est rapide : formatage et lint des fichiers modifiés. L'analyse statique complète et les tests, c'est le travail de la CI.

Dans la CI, sur chaque PR

C'est le filet de sécurité final, celui que personne ne peut contourner. Dans le workflow du chapitre CI/CD, vous avez déjà un job `lint` qui tourne en parallèle des tests. Il suffit d'y ajouter l'analyse statique :

```
lint:
  name: Style et analyse statique
  runs-on: ubuntu-latest
  steps:
    - uses: actions/checkout@v7
    - uses: shivammathur/setup-php@v2
      with: { php-version: "8.4", coverage: none }
    - run: composer install --no-interaction --prefer-dist --no-progress
    - run: vendor/bin/pint --test
    - run: vendor/bin/phpstan analyse --memory-limit=16
```

YAML

Même schéma pour Django (`ruff check .`, `ruff format --check .`, `mypy .`) et pour Node (`npx prettier --check .`, `npx eslint .`, `npx tsc --noEmit`). Une fois ce job dans la liste des checks obligatoires de votre protection de branche, plus aucune variable inutilisée ni aucun `null` non géré ne passe sur `main`.

Avec un assistant IA

Si vous codez avec un agent, ces outils sont ses meilleurs alliés. Un agent qui a accès à la sortie de PHPStan ou de `tsc` corrige ses propres erreurs de types avant que vous ne les voyiez.

Demandez-lui explicitement de lancer le lint et l'analyse statique après chaque modification, et ajoutez-le dans vos instructions de projet. C'est une des manières les plus efficaces de tenir la règle d'or du cours : **comprendre et valider chaque suggestion**. Ici, c'est la machine qui valide une partie du travail de la machine.

Les cousins : l'audit des dépendances

Une dernière famille d'outils lit non pas votre code, mais la liste de vos dépendances, et la compare aux bases de vulnérabilités connues. Une faille dans une bibliothèque que vous utilisez est une faille dans votre application, même si votre code est parfait.

Stack	Commande	Ce qu'elle fait
Laravel	<code>composer audit</code>	Compare <code>composer.lock</code> aux avis de sécurité Packagist
Django	<code>pip-audit</code> (à installer)	Compare les paquets installés à la base PyPI Advisory
Node	<code>npm audit</code>	Compare <code>package-lock.json</code> à la base GitHub Advisory

Ajoutez la commande de votre stack au job `lint`. Et activez **Dependabot** dans les réglages du repo (*Security* → *Dependabot alerts* et *security updates*) : GitHub ouvre alors une PR de mise à jour dès qu'une dépendance vulnérable est détectée, et votre CI vérifie que la mise à jour ne casse rien. C'est le même fichier `dependabot.yml` que celui du chapitre CI/CD, avec un `package-ecosystem` supplémentaire (`composer`, `pip` ou `npm`).

6. Adopter ces outils sur un projet existant



Si vous lancez PHPStan niveau 8 sur un projet de trois semaines, vous obtenez quatre cents erreurs et l'envie d'abandonner. Voici la méthode.

- 1. Le formateur d'abord, d'un seul coup.** Lancez-le sur tout le projet, committez le résultat seul dans un commit intitulé « Formatage », mergez. À partir de là, plus aucun diff de style ne polluera vos PR.

2. **Le linter avec les règles recommandées.** Corrigez ce qui est automatique (`--fix`), traitez le reste à la main, ça va vite. Ajoutez le job en CI.
3. **Excluez ce que vous n'avez pas écrit.** Migrations générées, `vendor/` , `node_modules/` , `.venv/` , dossiers de build : ils n'ont rien à faire dans l'analyse. C'est le rôle des clés `exclude` et `ignores` des configurations ci-dessus. Le reste, vous l'avez écrit, vous en répondez.
4. **L'analyse statique au niveau le plus bas qui passe.** PHPStan niveau 2, mypy sans `strict` , TypeScript sans `strict` . Si même ce niveau remonte trop d'erreurs, générez une *baseline* : un fichier qui liste les erreurs existantes et les ignore, pour que seules les **nouvelles** erreurs fassent échouer la CI. PHPStan le fait avec `--generate-baseline` ; mypy et tsc n'ont pas d'équivalent natif, on restreint alors les dossiers analysés.
5. **Montez d'un cran quand le projet est propre.** Chaque niveau supplémentaire est une session d'une heure à corriger des erreurs qui, souvent, sont de vrais bugs. C'est du temps rentable.
6. **Ne baissez jamais le niveau pour faire passer la CI.** Une erreur qu'on ne comprend pas est une erreur à comprendre, pas à cacher. Si c'est un vrai faux positif, ignorez cette ligne précise, avec un commentaire qui explique pourquoi.

ASTUCE

Sur un projet neuf comme votre Projet 1, faites tout ça **avant la première fonctionnalité**. Formateur, linter, analyseur au niveau 5, job CI : une heure au total, et vous ne connaîtrez jamais le problème des quatre cents erreurs.

7. À vous : TP

Sur votre projet, en binôme, 30 minutes :

1. **Formatez.** Installez le formateur de votre stack, lancez-le sur tout le projet, committez « Formatage » seul. Regardez la taille du diff : c'est tout le bruit que vous n'aurez plus jamais dans vos PR.
2. **Lintez.** Activez les règles recommandées, corrigez avec `--fix` , puis à la main. Notez la règle qui a remonté le plus d'erreurs et discutez-en : est-ce une mauvaise habitude à corriger ou une règle à désactiver pour votre projet, et pourquoi ?
3. **Analysez.** Installez l'analyseur statique au niveau le plus bas. Montez cran par cran jusqu'à ce qu'il remonte au moins une erreur. Lisez-la : est-ce un vrai bug ? Dans la majorité des cas, oui.
4. **Introduisez un bug de type volontairement.** Un `find()` sans vérification de `null` , une fonction appelée avec un argument du mauvais type. Lancez l'analyseur : il doit le voir. Lancez vos tests : les voient-ils ? Probablement pas. C'est toute la complémentarité.
5. **Branchez la CI.** Ajoutez les commandes au job `lint` de votre workflow, poussez, vérifiez le vert, ajoutez le job aux checks obligatoires de `main` .

Dans le rapport de projet, la section « qualité du code » gagne à contenir : les outils choisis et leur configuration, le niveau d'analyse statique atteint, et un exemple de bug attrapé par l'analyseur avant d'arriver en production.

En résumé

- **Trois questions, trois familles** : le formateur uniformise, le linter signale les motifs suspects, l'analyseur statique prouve des incohérences de types. On cumule les trois.
- **Sans exécuter le code** : l'analyseur lit la structure, déduit les types, vérifie chaque usage. Rapide, exhaustif, complémentaire des tests.
- **Plus vous typez, plus il voit** : annotez vos fonctions en PHP et en Python, activez `strict` en TypeScript.
- **Les outils** : Pint et PHPStan/Larastan ; ruff et mypy/django-stubs ; Prettier, ESLint et tsc/vue-tsc.
- **Un message d'outil = fichier, ligne, code de règle, phrase** : le code de règle mène à la doc, la doc dit pourquoi. On ne désactive rien qu'on n'a pas compris.
- **Trois moments** : dans l'éditeur (le plus utile), au commit (optionnel, rapide), en CI (obligatoire, dans les checks de branche).
- **L'audit des dépendances** (`composer audit`, `pip-audit`, `npm audit`) et Dependabot complètent le tableau : une faille dans une bibliothèque est une faille chez vous.
- **Adoption** : formater d'un coup, linter avec les règles recommandées, analyser au niveau bas et monter cran par cran. Jamais baisser le niveau pour faire passer la CI.

Pour aller plus loin

- Pint : <https://laravel.com/docs/pint> · PHPStan : <https://phpstan.org/user-guide/getting-started> · Larastan : <https://github.com/larastan/larastan>
- ruff : <https://docs.astral.sh/ruff> · mypy : <https://mypy.readthedocs.io> · django-stubs : <https://github.com/typeddjango/django-stubs>
- Prettier : <https://prettier.io/docs> · ESLint : <https://eslint.org/docs/latest/use/configure> · typescript-eslint : <https://typescript-eslint.io> · TypeScript `strict` : <https://www.typescriptlang.org/tsconfig/#strict>
- lint-staged : <https://github.com/lint-staged/lint-staged> · pre-commit : <https://pre-commit.com>
- Repo d'exemple du cours : <https://github.com/opmvpc/cicd26>

1.5 Où faire tourner votre application ? Du VPS au serverless

Point de théorie · à lire après avoir choisi votre hébergeur, pour comprendre ce que vous avez choisi

Slides du cours

-  **deck5-hebergement.pdf**  — les slides du point de théorie

Au premier cours, vous avez comparé des hébergeurs et vous en avez choisi un pour votre chat. Ce chapitre explique ce qui se cache derrière ce choix. Un hébergeur n'est pas qu'un prix et un quota : c'est un partage des tâches entre vous et lui. Selon la formule, vous installez tout vous-même ou vous ne touchez à rien, et chaque formule a ses bugs typiques. Les connaître vous évite de découvrir en semaine 5 que votre base de données a été effacée ou que vos WebSockets ne passeront jamais.

À la fin de ce chapitre, vous serez capables de :

1. **situer** un hébergeur sur l'échelle qui va du serveur nu aux fonctions à la demande, et dire ce que vous gérez vous-même à chaque marche ;
2. **décrire** ce qu'un serveur classique contient et pourquoi chaque couche est là ;
3. **expliquer** ce qu'est le serverless, comment une fonction vit et meurt, et pourquoi elle n'a pas de mémoire ;
4. **reconnaître** les trois pièges du serverless dans votre propre code ;
5. **justifier** le choix d'hébergement de votre projet avec des arguments techniques, dans votre rapport et à l'oral.

1. Une même application, quatre façons de la faire tourner

Prenez votre chat. Un backend en Laravel, Django ou Node, une base de données, des WebSockets pour le temps réel. Pour le rendre accessible sur une URL publique, il faut une machine quelque part qui exécute ce code. La question de l'hébergement, c'est : **qui s'occupe de cette machine, et jusqu'où ?**

Qui gère quoi ? L'échelle de responsabilité

La même application, quatre formules. Chaque couche est gérée soit par vous, soit par l'hébergeur.

☒ vous
 ☐ l'hébergeur

	VPS nu Hetzner · OVH · DigitalOcean	VPS + panneau Coolify · Dokploy installés sur le VPS	PaaS managé Render · Railway Fly.io · Laravel Cloud	Serverless Vercel · Netlify Cloudflare Workers
Votre code				
Scaling				automatique
Sauvegardes				
HTTPS et déploiement		le panneau		
Base de données		le panneau		
Runtime		le panneau		
Système et pare-feu				
Machine et réseau				

← plus de contrôle, plus de travail moins de travail, règles de la plateforme →

SIDEV · Projet d'intégration de développement L'échelle de responsabilité

Formule	Exemples	Ce que vous gérez	Ce que l'hébergeur gère
VPS nu	Hetzner, OVH, DigitalOcean	Tout : système, sécurité, runtime, base, HTTPS, déploiement, sauvegardes	La machine virtuelle et le réseau
VPS + panneau	Coolify ou Dokploy installé sur le VPS	Le VPS (mises à jour, sécurité) et vos réglages dans le panneau	Rien de plus, mais le panneau automatise déploiement, HTTPS, bases
PaaS managé	Render, Railway, Fly.io, Laravel Cloud	Votre code, vos variables d'environnement	Système, runtime, HTTPS, déploiement, base, sauvegardes
Serverless	Vercel, Netlify, Cloudflare Workers	Vos fonctions et vos pages	Tout le reste, y compris le nombre de machines

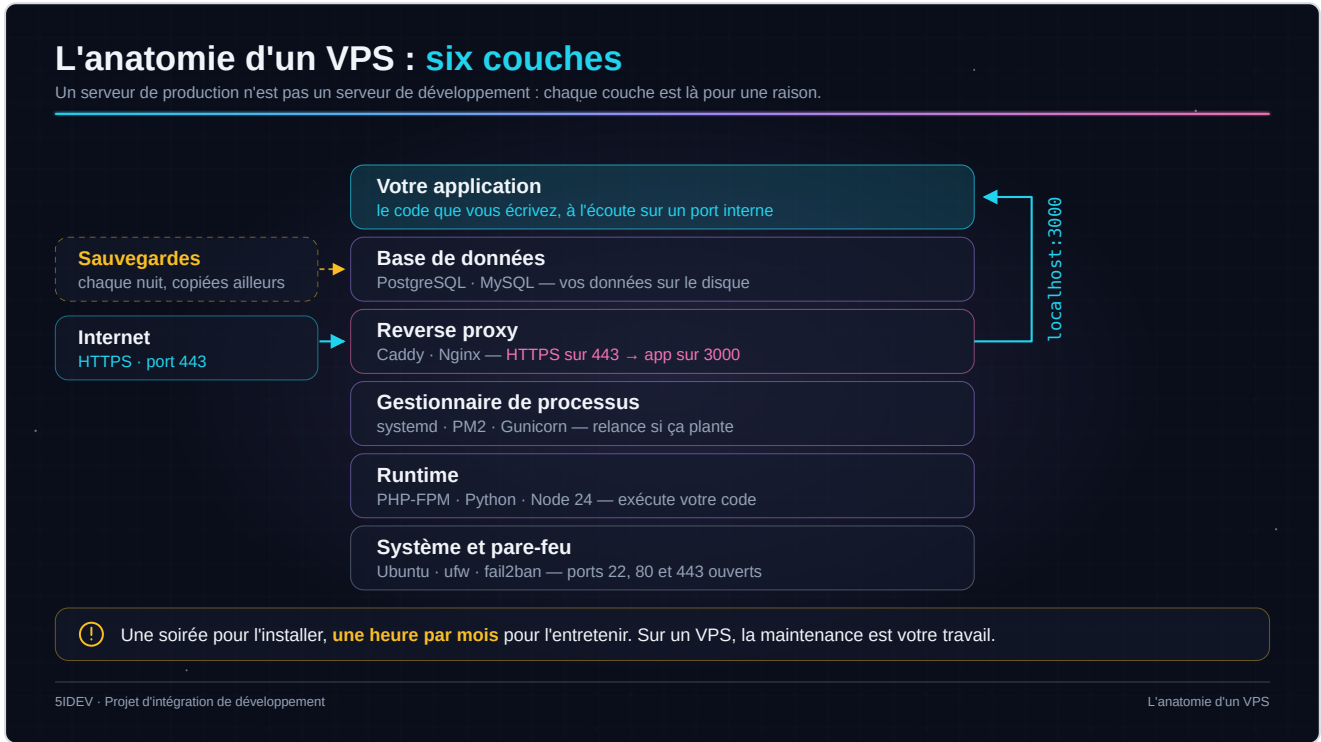
Un **VPS** (Virtual Private Server) est une machine virtuelle louée, allumée en permanence, sur laquelle vous avez tous les droits. Un **PaaS** (Platform as a Service) est un service qui prend votre code et le fait tourner sans que vous voyiez la machine. Le **serverless** va un cran plus loin : il n'y a plus d'application qui tourne en continu, seulement des fonctions lancées à chaque requête.

Plus vous montez dans le tableau, plus vous avez de contrôle et de travail. Plus vous descendez, moins vous avez de travail et plus vous acceptez les règles de la plateforme. Il n'y a pas de bonne marche. Il y a la marche qui correspond à votre projet, à votre budget et à vos compétences. La suite du chapitre parcourt l'échelle de haut en bas.

2. Le VPS de l'intérieur

Ce qu'il y a dans un serveur

Sur votre PC, `php artisan serve`, `python manage.py runserver` ou `node server.js` suffit. Sur un serveur public, ce serveur de développement ne convient plus : il n'est pas prévu pour tenir la charge, il ne fait pas de HTTPS et il meurt avec votre terminal. Un serveur de production empile six couches.



Couche	Exemples	Ce qu'elle fait	Sans elle
Système et pare-feu	Ubuntu, <code>ufw</code> , <code>fail2ban</code>	Ferme tous les ports sauf 22, 80 et 443, bloque les tentatives de connexion répétées	Un serveur exposé sans pare-feu reçoit des attaques par mot de passe dès la première heure
Runtime	PHP-FPM, Python, Node 24	Exécute votre code	Rien ne tourne
Gestionnaire de processus	<code>systemd</code> , PM2, Gunicorn	Lance l'application au démarrage, la relance si elle plante	Un crash à 3h du matin, et le site reste mort jusqu'à ce que quelqu'un le remarque
Reverse proxy	Caddy, Nginx	Reçoit le trafic sur les ports 80 et 443, gère le certificat HTTPS, le transmet à votre application sur son port interne	Pas de HTTPS, une application exposée directement

Couche	Exemples	Ce qu'elle fait	Sans elle
Base de données	MySQL, PostgreSQL	Stocke vos données sur le disque	Vos messages disparaissent au redémarrage
Sauvegardes	<code>mysqldump</code> + un cron, ou un outil dédié	Copie la base ailleurs chaque nuit	Une mauvaise commande, et sept semaines de données sont perdues

Un **reverse proxy** est un serveur qui se place devant votre application. Le navigateur lui parle en HTTPS sur le port 443, et lui parle à votre application en HTTP sur un port interne, par exemple 3000. C'est lui qui porte le certificat.

À quoi ressemble l'installation

Voici le strict minimum pour une application Node sur un VPS Ubuntu neuf. Chaque ligne correspond à une couche du tableau.

BASH

```
# Système à jour et pare-feu : seuls SSH, HTTP et HTTPS passent
sudo apt update && sudo apt upgrade -y
sudo ufw allow 22 && sudo ufw allow 80 && sudo ufw allow 443
sudo ufw enable

# Runtime : Node 24 LTS
curl -fsSL https://deb.nodesource.com/setup_24.x | sudo -E bash -
sudo apt install -y nodejs

# Base de données
sudo apt install -y postgresql

# Reverse proxy avec HTTPS automatique
sudo apt install -y caddy
```

Le fichier `/etc/caddy/Caddyfile` tient en trois lignes. Caddy obtient et renouvelle le certificat HTTPS tout seul, ce qui explique qu'on le préfère à Nginx pour un premier serveur :

```
chat.exemple.be {
    reverse_proxy localhost:3000
}
```

Reste à lancer l'application et à la faire survivre aux redémarrages. Un service `systemd` décrit comment la démarrer, avec quel utilisateur et quoi faire si elle plante :

```
# /etc/systemd/system/chat.service
[Unit]
Description=Chat 5IDEV
After=network.target postgresql.service

[Service]
User=chat
WorkingDirectory=/srv/chat
EnvironmentFile=/srv/chat/.env
ExecStart=/usr/bin/node server.js
Restart=always

[Install]
WantedBy=multi-user.target
```

BASH

```
sudo systemctl enable --now chat
```

Pour Laravel, remplacez le runtime par PHP-FPM et le service par une configuration Caddy qui exécute PHP ; pour Django, le service lance Gunicorn au lieu de `node`. Le schéma est le même : un pare-feu, un runtime, un processus surveillé, un proxy, une base.

Ce que ce script ne montre pas, c'est le temps. Comptez une soirée pour le faire marcher la première fois, et une heure par mois ensuite : mises à jour de sécurité, renouvellement d'une clé SSH, disque plein à cause des logs, mise à jour de Node. Sur un VPS, **la maintenance est votre travail**, et personne ne vous préviendra si vous l'oubliez.

Ce que ça vous donne

En échange de ce travail, le VPS vous donne tout ce que les autres formules limitent :

- **Un processus qui reste allumé.** Une variable en mémoire garde sa valeur d'une requête à l'autre. Une connexion WebSocket reste ouverte des heures. Une tâche peut tourner toute la nuit.
- **Un disque.** Vous écrivez des fichiers où vous voulez, la base est à côté de l'application, sans limite de taille autre que le disque.
- **Un prix fixe.** Entre 4 et 10 € par mois pour une petite machine, quel que soit le trafic. Zéro surprise sur la facture.
- **Le contrôle.** Vous pouvez installer n'importe quoi, lire n'importe quel log, brancher n'importe quel outil.

Pour un chat en temps réel, ce sont exactement les propriétés dont vous avez besoin. Retenez-le pour la partie 5.

Coolify et Dokploy : le VPS avec le confort d'un PaaS

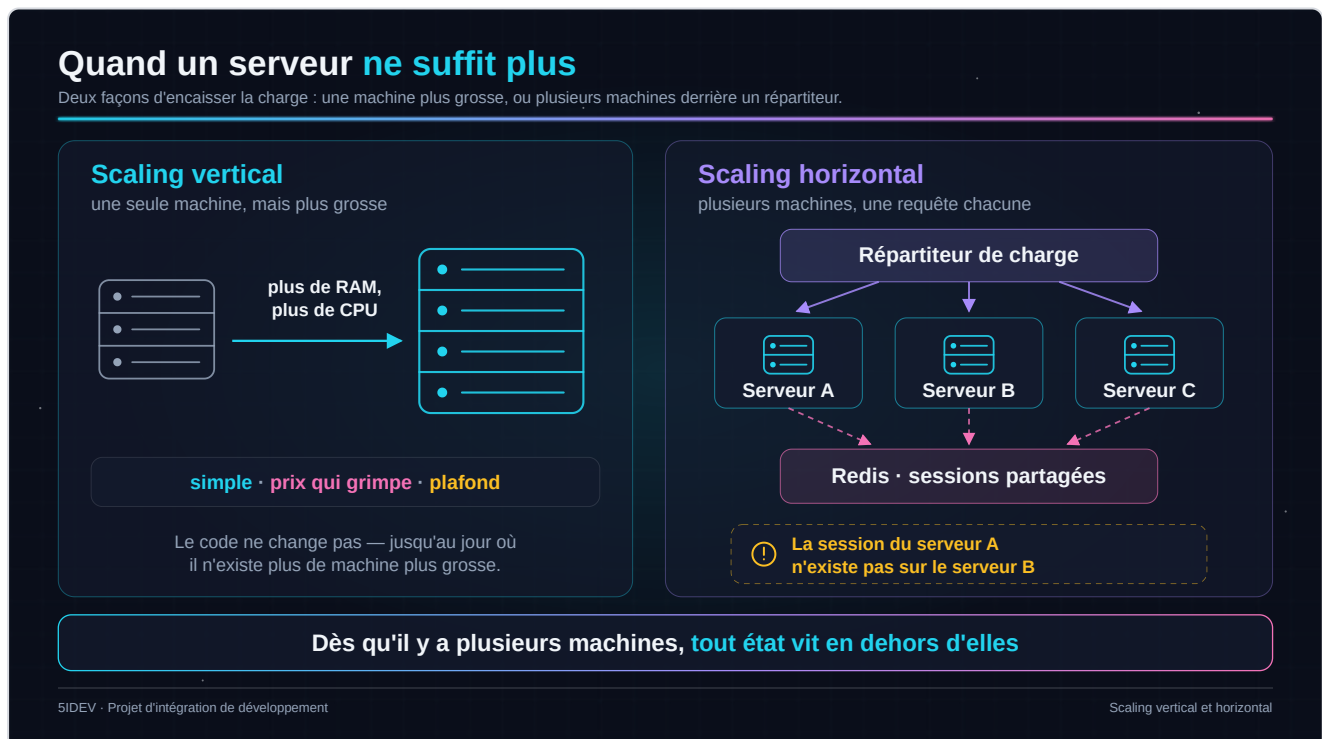
Le chapitre CI/CD vous l'a présenté : sur un VPS, vous pouvez installer un **panneau de déploiement** comme Coolify ou Dokploy. C'est une application web qui tourne sur votre serveur et qui fait pour vous ce que le script ci-dessus fait à la main. Vous lui donnez l'adresse de votre dépôt GitHub, elle détecte votre stack, construit l'application dans un conteneur, obtient le certificat HTTPS, crée une base de données et redéploie à chaque push sur `main`.

Un **conteneur** est un paquet qui contient votre application et tout ce dont elle a besoin pour tourner, isolé du reste du serveur. C'est l'unité de déploiement de Coolify, de Dokploy, mais aussi de Fly.io ou de Cloud Run : la même image tourne partout de la même façon.

Ce que le panneau ne fait pas : il ne met pas à jour le système, il ne surveille pas le disque, il ne vous protège pas d'une mauvaise commande. La machine reste la vôtre. Vous gardez le prix fixe et le contrôle, vous perdez la moitié de la maintenance.

3. Quand un serveur ne suffit plus

Imaginez que votre chat marche et que le trafic monte. Un petit VPS tient sans effort quelques centaines d'utilisateurs simultanés sur une application bien écrite, et une vingtaine sur une application qui fait cinquante requêtes SQL par page. Le jour où la machine sature, deux voies s'ouvrent.



Le **scaling vertical** consiste à louer une machine plus grosse. Plus de mémoire, plus de processeurs, même configuration. C'est simple et sans changement dans le code, mais le prix grimpe vite et il existe un plafond : à un moment, il n'y a plus de machine plus grosse.

Le **scaling horizontal** consiste à multiplier les machines et à placer devant elles un **répartiteur de charge** (load balancer), qui distribue chaque requête à l'un des serveurs et écarte ceux qui ne répondent plus. Si un serveur tombe, le site reste en ligne. Mais le code doit changer, parce que deux requêtes du même utilisateur peuvent atterrir sur deux machines différentes.

Le premier symptôme, ce sont les sessions. Une session stockée en mémoire sur le serveur A n'existe pas sur le serveur B : l'utilisateur est déconnecté une requête sur deux. La solution est de sortir la session de la machine, dans un magasin partagé comme Redis, une base clé-valeur en mémoire :

JAVASCRIPT

```
// Session en mémoire : ne survit pas au passage sur un autre serveur
app.use(session({ store: new MemoryStore() }));

// Session dans Redis : la même pour tous les serveurs
app.use(session({ store: new RedisStore({ client: redis }) }));
```

Le même raisonnement vaut pour les fichiers envoyés par les utilisateurs, pour le cache et pour les WebSockets, dont les connexions se répartissent elles aussi entre serveurs. **Dès qu'il y a plusieurs machines, tout état doit vivre en dehors d'elles.** Retenez cette phrase : c'est aussi la règle du serverless, poussée à l'extrême.

Le scaling horizontal a un coût : trois serveurs, un répartiteur, un Redis, et les compétences pour faire tenir l'ensemble. La plupart des équipes ne veulent pas de ce travail. C'est précisément ce que les PaaS et le serverless vendent : le scaling sans l'infrastructure.

4. Le serverless, vraiment

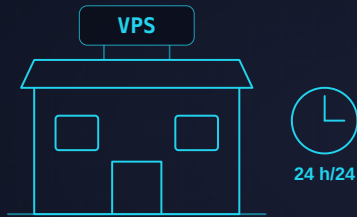
Un serveur qui n'existe qu'au moment de la requête

Le mot est trompeur. Il y a toujours des serveurs, mais ce ne sont plus les vôtres, et votre application ne tourne plus en continu dessus. Vous déployez des **fonctions**. Quand une requête arrive, la plateforme démarre une instance de la fonction, lui passe la requête, renvoie la réponse, puis détruit l'instance au bout de quelques minutes sans trafic. Vous payez le temps d'exécution, pas le temps d'allumage.

Serveur allumé 24 h/24 ou fonction à la demande

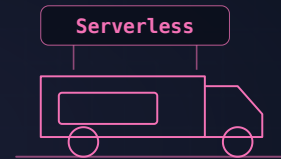
Le serverless ne supprime pas les serveurs : il supprime le serveur qui vous attend, allumé, sans rien faire.

Le restaurant



La cuisine tourne jour et nuit.
Vous payez même à 4 h du matin, salle vide.

Le food truck



4 h aucun camion

12 h dix camions

Il vient à la commande et repart.
Vous payez le temps d'exécution.

Mille utilisateurs à midi : **mille instances**, sans rien configurer.

5IDEV - Projet d'intégration de développement

Restaurant ou food truck

Un VPS est un restaurant dont la cuisine tourne jour et nuit. Vous payez le cuisinier même à 4 h du matin, quand la salle est vide. Le serverless est un food truck : il ne vient que quand quelqu'un commande, et il repart ensuite. À midi, la plateforme envoie dix food trucks ; à 4 h, aucun. C'est ce que fait Vercel quand mille utilisateurs arrivent en même temps : mille instances, sans que vous ayez rien configuré.

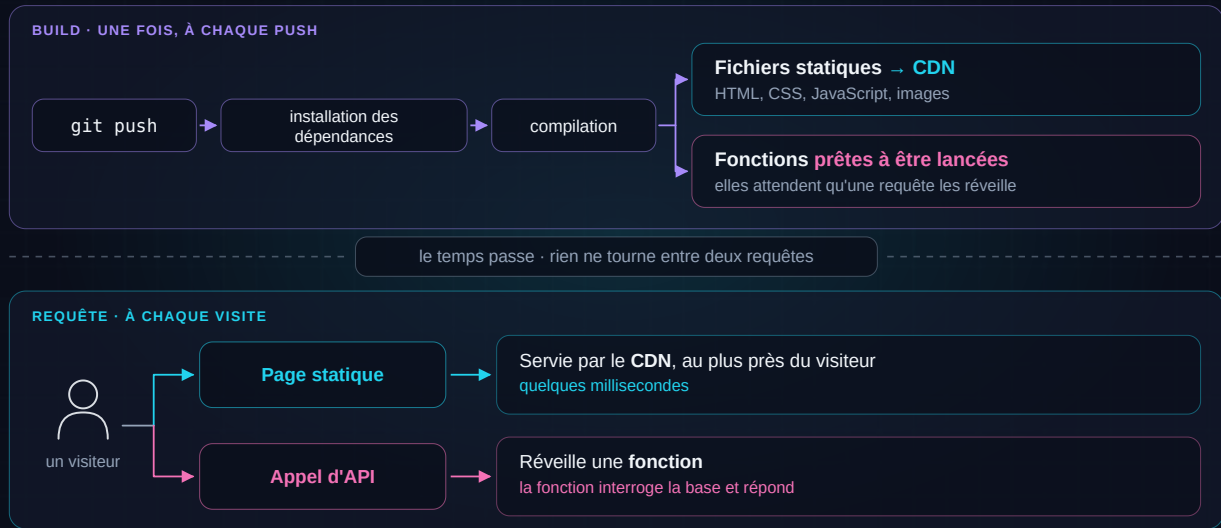
Dans ce chapitre, Vercel sert d'exemple parce que c'est la plateforme que vous rencontrerez le plus souvent avec Next.js, Nuxt ou SvelteKit. Netlify fonctionne de la même façon. Cloudflare Workers pousse l'idée plus loin, avec des fonctions plus légères qui démarrent en quelques millisecondes dans des centaines de villes. Le modèle est le même partout.

Deux temps : le build et la requête

Sur un VPS, vous lancez l'application et elle attend les requêtes. En serverless, le déploiement se coupe en deux moments bien distincts.

Deux temps : le **build**, puis chaque **requête**

Le build prépare tout une fois ; ensuite, seule la visite déclenche quelque chose.



S1DEV · Projet d'intégration de développement

Build et requête

Le **build** se produit une fois, à chaque push. La plateforme installe les dépendances, compile le projet, produit d'un côté des fichiers statiques (HTML, CSS, JavaScript, images) envoyés sur un **CDN**, un réseau de serveurs répartis dans le monde qui servent ces fichiers au plus près du visiteur, et de l'autre des fonctions prêtes à être lancées. Une page qui ne dépend d'aucune donnée est construite ici, une fois pour toutes.

La **requête** se produit à chaque visite. Une page statique vient du CDN en quelques millisecondes. Un appel d'API réveille une fonction, qui interroge la base et répond. Rien ne tourne entre deux requêtes.

À quoi ressemble une fonction

Voici la route qui liste et crée les messages d'une conversation, en Next.js. Le fichier `app/api/messages/route.js` devient une fonction serverless au déploiement sur Vercel :

JAVASCRIPT

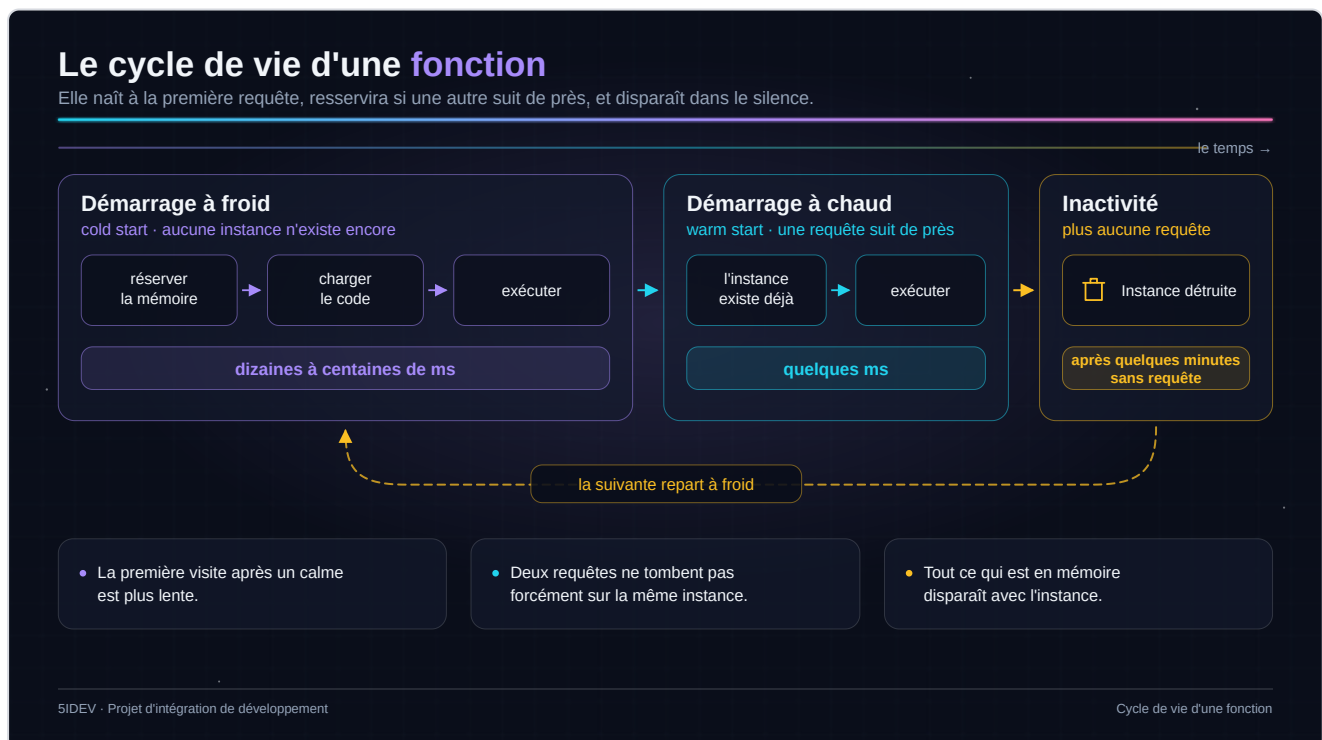
```
import { sql } from "../db"; // client de base de données, voir plus bas

// Chaque export nommé d'après une méthode HTTP devient un point d'entrée
export async function GET() {
  const messages = await sql`SELECT * FROM messages ORDER BY id DESC LIMIT 50`;
  return Response.json(messages);
}

export async function POST(request) {
  const { body } = await request.json();
  if (!body?.trim()) {
    return Response.json({ error: "Le message ne peut pas être vide" }, { status: 422 });
  }
  const [message] = await sql`INSERT INTO messages (body) VALUES (${body}) RETURNING *`;
  return Response.json(message, { status: 201 });
}
```

Comparez avec le serveur Express du chapitre CI/CD : pas de `app.listen()`, pas de port, pas de processus qui attend. Le fichier exporte des fonctions, et la plateforme décide quand les appeler. Vercel accepte aussi des fonctions Python (`api/messages.py`, avec Flask ou FastAPI) et, pour Laravel, l'équivalent s'appelle Laravel Cloud : votre application dort quand personne ne l'utilise et se réveille à la première requête, base de données comprise.

Le cycle de vie : cold start, warm start, disparition



Quand une requête arrive et qu'aucune instance n'existe, la plateforme en crée une : elle réserve de la mémoire, charge votre code et ses dépendances, puis exécute la fonction. C'est le **démarrage à froid** (cold start), et il coûte entre quelques dizaines et quelques centaines de millisecondes selon la taille de votre code. Si une deuxième requête arrive juste après, l'instance est encore là et répond en quelques millisecondes : **démarrage à chaud** (warm start). Après quelques minutes sans requête, l'instance est détruite. La suivante repartira à froid.

Trois conséquences pratiques :

- La première visite après une période calme est plus lente. Un site peu fréquenté a un cold start sur presque chaque visite.
- Vous ne pouvez pas savoir si deux requêtes tomberont sur la même instance. Deux visiteurs simultanés peuvent avoir deux instances, ou une seule.
- Tout ce que vous mettez en mémoire dans une instance disparaît avec elle.

Les limites, en chiffres

Une fonction n'est pas un serveur allégé. Elle vit dans un cadre strict, que la plateforme impose pour pouvoir en lancer des milliers en parallèle. Chez Vercel, en septembre 2026 :

Limite	Plan gratuit (Hobby)	Plan Pro	Ce que ça change pour vous
Durée maximale d'une exécution	5 minutes	13 minutes	Un export PDF de 20 minutes est impossible dans une fonction
Mémoire	2 Go	jusqu'à 4 Go	Une conversion vidéo ne rentre pas
Corps d'une requête ou d'une réponse	4,5 Mo	4,5 Mo	Un upload de photo passe, une vidéo non
Disque	/tmp seulement, effacé avec l'instance	idem	Rien ne s'écrit durablement sur le disque
Facturation	gratuit dans les quotas	au temps de CPU actif et à la mémoire réservée	Le prix suit le trafic, dans les deux sens

Ces chiffres changent chaque année, toujours à la hausse. Il y a deux ans, la limite gratuite était de 10 secondes. Ce qui ne change pas, c'est la nature de ces limites : une fonction est faite pour répondre vite et disparaître.

5. Les trois pièges du serverless

Les trois erreurs qui suivent viennent toutes du même réflexe : écrire du code de VPS dans une fonction. Chacune a sa solution, et cette solution est toujours la même que celle du scaling horizontal : **sortir l'état de la machine**.

Piège n°1 : garder quelque chose en mémoire

JAVASCRIPT

```
let visites = 0; // vit dans l'instance, pas dans l'application

export async function GET() {
  visites++;
  return Response.json({ visites });
}
```

Sur un VPS, ce compteur affiche 1, 2, 3, 4. Sur Vercel, il affiche 1, 2, 1, 3, 1, 1. Chaque nouvelle instance repart de zéro, et plusieurs instances vivent en parallèle. Le même problème touche un cache maison, une liste d'utilisateurs connectés, une session stockée dans une variable.



La solution est d'écrire l'état dans un service qui survit aux instances. Pour un compteur ou une session, Redis. Vercel ne propose plus son propre Redis depuis fin 2024 : on en installe un depuis sa place de marché, chez Upstash par exemple, et on l'appelle en HTTP :

JAVASCRIPT

```
import { Redis } from "@upstash/redis";

const redis = Redis.fromEnv(); // lit l'URL et le jeton dans les variables
d'environnement

export async function GET() {
  const visites = await redis.incr("visites");
  return Response.json({ visites });
}
```

ATTENTION

Une connexion à une base de données est aussi un état. Sur un VPS, votre application ouvre dix connexions au démarrage et les réutilise. En serverless, mille instances ouvriraient mille connexions et feraient tomber la base. Les bases pensées pour le serverless (Neon, PlanetScale, Supabase) placent un **pooler** devant elles, un intermédiaire qui partage un petit nombre de connexions réelles entre toutes les fonctions. Utilisez l'URL de connexion « pooled » qu'elles fournissent, jamais l'URL directe.

Piège n°2 : écrire un fichier sur le disque

JAVASCRIPT

```
import { writeFile } from "node:fs/promises";

export async function POST(request) {
  const photo = await request.arrayBuffer();
  await writeFile("./uploads/photo.jpg", Buffer.from(photo)); // ne persiste pas
  return Response.json({ ok: true });
}
```

L'écriture réussit peut-être dans `/tmp`, puis le fichier disparaît avec l'instance, et une autre instance ne le verra jamais. Les fichiers envoyés par vos utilisateurs vont dans un **stockage d'objets** : Vercel Blob, Cloudflare R2, Amazon S3. Vous y envoyez le fichier, vous recevez une URL, vous stockez l'URL en base :

JAVASCRIPT

```
import { put } from "@vercel/blob";

export async function POST(request) {
  const photo = await request.arrayBuffer();
  const blob = await put("photos/photo.jpg", photo, { access: "public" });
  return Response.json({ url: blob.url });
}
```

Piège n°3 : lancer un traitement long

Convertir une vidéo, générer un PDF de mille pages, envoyer dix mille e-mails : tout ce qui dépasse la durée maximale d'une fonction est coupé net, avec une erreur 504 pour l'utilisateur. Et même sous la limite, faire attendre un visiteur trois minutes sur une requête HTTP est une mauvaise idée sur n'importe quel hébergeur.



La solution est une **file de tâches**. La fonction reçoit la demande, dépose une tâche dans la file et répond tout de suite « c'est en cours ». Un **worker**, un programme qui reste allumé sur un VPS ou chez un service dédié, prend les tâches une par une et fait le travail, sans limite de durée. Quand il a fini, il prévient l'application. Avec Trigger.dev, un service de files pensé pour Vercel, la tâche s'écrit comme une fonction ordinaire :

JAVASCRIPT

```

// trigger/transcode.js : tourne chez Trigger.dev, pas dans la fonction Vercel
import { task } from "@trigger.dev/sdk";

export const transcode = task({
  id: "transcode-video",
  run: async ({ videoUrl }) => {
    await convertir(videoUrl); // vingt minutes, personne n'attend
  },
});
  
```

JAVASCRIPT

```
// app/api/videos/route.js : la fonction Vercel ne fait que déposer la tâche
import { transcode } from "@trigger/transcode";

export async function POST(request) {
  const { videoUrl } = await request.json();
  const run = await transcode.trigger({ videoUrl });
  return Response.json({ runId: run.id, status: "en cours" }, { status: 202 });
}
```

Laravel a ses queues et ses workers depuis toujours, Django a Celery : si vous êtes sur un VPS ou un PaaS, vous avez déjà ce mécanisme sous la main, et la même règle s'applique. Un traitement long ne se fait jamais dans la requête.

NOTE

Les WebSockets sont un quatrième cas du même problème. Une connexion WebSocket reste ouverte pendant des heures, et une fonction est faite pour vivre quelques secondes. Vercel expérimente un support depuis 2026, mais la pratique courante reste de confier le temps réel à un service dédié (Pusher, Ably) ou à un serveur qui reste allumé. Pour un chat, c'est la question qui décide de tout.

6. Choisir, pour de vrai

Quatre questions

Vous avez maintenant tout ce qu'il faut pour relire la grille d'hébergement que vous avez remplie au premier cours. Quatre questions suffisent à écarter les mauvaises options.



- 1. Votre application garde-t-elle une connexion ouverte ?** WebSockets, flux en continu, jeu en ligne. Si oui, il vous faut un processus qui reste allumé : VPS, avec ou sans panneau, ou un PaaS qui accepte les connexions longues. Le serverless pur est écarté, sauf à déléguer le temps réel à un service tiers.
- 2. Avez-vous des traitements longs ou des fichiers à écrire ?** Si oui, prévoyez un worker et un stockage d'objets, quelle que soit la formule. Sur un VPS, c'est votre disque et un worker à côté. En serverless, c'est une file de tâches et un Blob.
- 3. Votre trafic est-il prévisible ?** Un projet étudiant, un outil interne, un site à trafic constant : le prix fixe du VPS ou d'un petit PaaS est imbattable. Un lancement public avec des pics inconnus : le serverless absorbe les pics sans que vous ayez à dimensionner quoi que ce soit, et la facture suit.
- 4. Qui fera la maintenance ?** Si personne dans l'équipe n'a envie de passer une heure par mois dans un terminal SSH, prenez un PaaS ou un VPS avec Coolify. Un VPS nu sans maintenance devient une machine compromise en quelques mois.

Par type de projet

Projet	Formule qui convient	Pourquoi
Site vitrine, blog, documentation	Serverless ou hébergement statique	Pages construites au build, servies par le CDN, coût nul
Application web classique, trafic modéré	PaaS, ou VPS avec Coolify	Un processus allumé, un prix fixe, peu de maintenance
Chat, jeu, tableau collaboratif en temps réel	VPS ou PaaS avec connexions longues	Les WebSockets ont besoin d'un processus qui dure

Projet	Formule qui convient	Pourquoi
API publique aux pics imprévisibles	Serverless	Mille instances à midi, zéro la nuit, sans configuration
Traitements lourds : vidéo, PDF, données	Hybride : serverless ou PaaS devant, workers derrière	La requête reste courte, le travail se fait ailleurs
Application interne, données sensibles	VPS	Contrôle total sur la machine et sur l'emplacement des données

Ce que ça donne pour votre chat

Votre Projet 1 coche la case du temps réel. C'est pour cela que l'activité du premier cours vous demandait de vérifier les WebSockets et la persistance de la base avant tout le reste. Un groupe sur Render ou Railway a un processus allumé et peut ouvrir des WebSockets ; il doit surveiller la mise en veille du plan gratuit, qui endort l'application après un quart d'heure sans trafic et la réveille en une bonne minute, et la durée de vie de la base gratuite. Un groupe sur un VPS avec Coolify a tout ce qu'il faut et paie un prix fixe, en échange d'une machine à tenir à jour. Un groupe sur Vercel peut y mettre le front et l'API, mais doit confier le temps réel à un service tiers ou à un second serveur.

Dans votre rapport, la section « hébergement » attend exactement ce raisonnement : la formule choisie, la marche de l'échelle où elle se situe, ce que vous gérez vous-même, et ce qui vous a fait écarter les autres. Trois phrases par question du point précédent suffisent.

En résumé

- **Héberger, c'est partager le travail.** Du VPS nu au serverless, chaque marche déplace des tâches de vous vers la plateforme, et du contrôle dans l'autre sens.
- **Un VPS est un processus qui reste allumé,** avec un disque et un prix fixe. Il donne tout, y compris la maintenance. Coolify ou Dokploy en enlève la moitié.
- **Une fonction serverless naît à la requête et meurt après.** Elle démarre à froid, ne garde rien en mémoire, n'écrit pas sur le disque et s'arrête après quelques minutes.
- **Tout état vit en dehors de la machine :** Redis pour les sessions et les compteurs, un stockage d'objets pour les fichiers, une file et un worker pour les traitements longs, un pooler devant la base. C'est la règle du scaling horizontal, et le serverless l'impose dès la première requête.
- **Les WebSockets décident.** Une connexion longue demande un processus qui dure. Pour un chat, cette question passe avant le prix.

Pour aller plus loin

- Limites des fonctions Vercel : <https://vercel.com/docs/functions/limitations>

- Route Handlers Next.js : <https://nextjs.org/docs/app/api-reference/file-conventions/route>
- Redis sur Vercel via Upstash : <https://vercel.com/docs/redis> · Vercel Blob : <https://vercel.com/docs/vercel-blob>
- Trigger.dev, tâches en arrière-plan : <https://trigger.dev/docs>
- Laravel Cloud : <https://cloud.laravel.com/docs> · Cloudflare Workers : <https://developers.cloudflare.com/workers>
- Coolify : <https://coolify.io> · Dokploy : <https://dokploy.com> · Caddy : <https://caddyserver.com/docs>
- Le plan gratuit de Render et sa mise en veille : <https://render.com/docs/free>

1.6 Backups : sauvegarder et restaurer vos données

Point de théorie · à lire avant que votre application ne contienne des données auxquelles vous tenez

Slides du cours

-  Les slides du point de théorie (PDF) 

Le chapitre sur l'hébergement citait les sauvegardes dans une seule ligne de tableau : « une mauvaise commande, et sept semaines de données sont perdues ». Ce chapitre développe cette ligne. Il explique les principes, présente les outils par stack et par hébergeur, et renvoie à leur documentation officielle. Dans les parties 6 et 7, lisez ce qui correspond à votre projet.

À la fin de ce chapitre, vous serez capables de :

1. **lister** ce qui doit être sauvegardé dans une application web, et ce qui ne doit pas l'être ;
2. **expliquer** la règle 3-2-1 et les deux questions qui dimensionnent une stratégie de backup ;
3. **distinguer** un backup complet, différentiel et incrémental, un instantané et une restauration à un instant donné ;
4. **décrire** les cinq étapes d'un backup automatique et **choisir** l'outil qui correspond à votre stack et à votre hébergeur ;
5. **vérifier** qu'un backup se restaure, et savoir combien de temps cela prend.

1. Ce qui arrive aux données

Le 31 janvier 2017, tard le soir, un ingénieur de GitLab veut vider le dossier de données d'un serveur de base de données secondaire. Il se croit sur le serveur secondaire, il est sur la production. Quand il s'en rend compte et arrête la commande, environ 300 Go de la base ont disparu.

L'équipe se tourne alors vers ses sauvegardes. Sur le papier, GitLab en avait plusieurs : un export chaque nuit, des copies de disque, une base répliquée. L'export nocturne échouait en silence, parce que l'outil `pg_dump` installé était en version 9.2 et la base en version 9.6. Les e-mails d'alerte étaient rejetés par le serveur de messagerie avant d'arriver. Les copies de disque n'étaient pas activées sur les serveurs de base de données, et la réplication était en panne. Le site a été sauvé par une copie du disque qu'un ingénieur avait faite à la main six heures plus tôt, pour alimenter un serveur de test. Selon le rapport publié par GitLab, les modifications faites entre 17 h 20 et minuit (UTC) ont été perdues, soit environ 5 000 projets, 5 000 commentaires et 700 nouveaux comptes. Le site est resté à l'arrêt 18 heures.

Les données ne se perdent pas seulement quand un disque tombe en panne.

Cause	Exemple	Ce qui vous sauve
Erreur humaine	Un <code>DROP TABLE</code> ou un <code>migrate:fresh</code> lancé sur la production	Un backup de la veille
Bug dans le code	Une migration qui vide une colonne, un <code>UPDATE</code> sans <code>WHERE</code>	Un backup d'avant le déploiement
Panne ou sinistre	En mars 2021, un centre de données d'OVH brûle à Strasbourg	Une copie dans un autre lieu
Règle de l'hébergeur	La base gratuite de Render devient inaccessible après 30 jours, puis elle est supprimée	Une copie chez vous
Piratage	Un rançongiciel (<i>ransomware</i>) chiffre le serveur et tout ce qu'il peut atteindre, puis réclame une rançon	Une copie que le serveur ne peut pas effacer

Lors de l'incendie d'OVH, des clients avaient bien des sauvegardes, mais sur le même site que leurs serveurs.

ATTENTION

Trois choses ressemblent à un backup et n'en sont pas. Une **base répliquée** recopie chaque écriture sur un second serveur, y compris le `DROP TABLE`. Des **disques en RAID**, qui écrivent chaque donnée sur deux disques, protègent d'une panne de disque, pas d'une mauvaise commande. Un **dossier synchronisé** propage la suppression en quelques secondes. Un backup est une copie figée à une date, que l'application ne peut plus modifier.

2. Quoi sauvegarder

Une application web en production, c'est quatre choses. Chacune a sa propre copie de sécurité.

Quatre choses à ne pas perdre

Le code

copie : Git et GitHub, déjà fait

La base de données

copie : un export chaque nuit

Les fichiers envoyés

copie : une archive ou un second bucket

Les secrets (.env)

copie : un gestionnaire de mots de passe

Élément	Où il vit	Sa sauvegarde
Le code	Votre dépôt Git	GitHub et le clone de chaque membre du groupe. C'est déjà fait.
La base de données	PostgreSQL ou MySQL, sur le serveur ou chez un fournisseur	Un export : un fichier qui contient ce qu'il faut pour recréer les tables et leurs lignes. En anglais, un <i>dump</i> .
Les fichiers envoyés	Avatars, pièces jointes, PDF générés, sur le disque du serveur ou dans un stockage d'objets	Une archive du dossier, ou une copie dans un second bucket . Un bucket est un dossier dans un stockage d'objets comme Cloudflare R2 ou Amazon S3.
Les secrets	Le .env de production, les clés d'API	Un gestionnaire de mots de passe partagé par le groupe. Jamais dans Git.

Le reste se reconstruit et ne se sauvegarde pas : `vendor/` , `node_modules/` , le cache, les fichiers compilés, les logs. Un `composer install` ou un `npm ci` les recrée à partir du dépôt.

La base et les fichiers vont ensemble. Si la table `messages` contient le chemin `uploads/photo-42.jpg` et que le fichier a disparu, la restauration de la base seule donne une application pleine d'images cassées.

3. Les règles d'une bonne stratégie

La règle 3-2-1



La règle tient en trois chiffres. **Trois copies** des données : l'original et deux sauvegardes. **Deux supports** différents, par exemple le disque du serveur et un stockage d'objets. **Une copie hors site**, c'est-à-dire ailleurs que chez l'hébergeur de l'application.

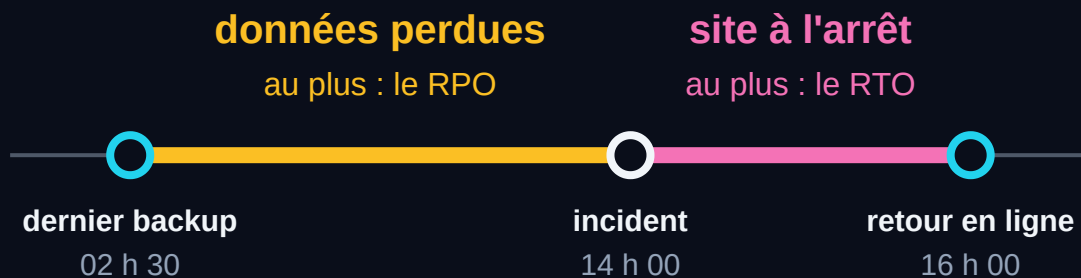
Pour votre projet, cela donne : la base en production, un export gardé sur le serveur pour restaurer vite, et le même export envoyé dans un bucket chez un autre fournisseur. Cloudflare R2, Backblaze B2 et les autres stockages d'objets proposent tous une API compatible avec celle d'Amazon S3. Un outil qui sait écrire dans « un bucket S3 » sait donc écrire chez n'importe lequel d'entre eux.

Deux questions pour régler la fréquence

Un backup est fait à 2 h 30, un incident survient à 14 h : tout ce qui a été écrit entre les deux est perdu. Avec un backup par nuit, cette perte peut atteindre 24 heures, si l'incident tombe juste avant le backup suivant. La perte maximale que vous acceptez s'appelle le **RPO**, pour *Recovery Point Objective* : l'objectif de point de reprise, c'est-à-dire le moment le plus ancien auquel vous acceptez de revenir. Un backup par nuit donne un RPO de 24 heures, un backup par heure un RPO d'une heure.

Entre l'incident et le retour en ligne, le site est à l'arrêt. L'arrêt maximal que vous acceptez s'appelle le **RTO**, pour *Recovery Time Objective* : l'objectif de délai de reprise. La durée réelle dépend de la taille du backup, de l'endroit où il se trouve, et surtout du fait que quelqu'un ait déjà fait la restauration une fois.

Ce que coûte un incident



RPO : la perte maximale que vous acceptez

RTO : l'arrêt maximal que vous acceptez

Pour un chat étudiant, perdre une journée de messages est acceptable : un backup par nuit suffit. Pour une boutique en ligne, perdre une journée de commandes ne l'est pas. Ces deux chiffres se décident avec le client, et ils fixent le prix de la solution.

Trois réglages de plus

- **La rétention** est la durée pendant laquelle vous gardez les anciens backups. Un bug qui abîme les données peut passer inaperçu une semaine. Si vous ne gardez que le backup d'hier, il contient déjà les données abîmées. Un réglage courant : 7 backups quotidiens, 4 hebdomadaires, 6 mensuels.
- **La confidentialité.** Un export contient les adresses e-mail, les mots de passe hachés et les messages privés de vos utilisateurs. Le bucket reste privé, et l'export n'entre jamais dans Git ni dans un dossier public du serveur. La plupart des outils savent aussi chiffrer le backup avec un mot de passe. Rangez ce mot de passe dans le gestionnaire du groupe : sans lui, le backup est illisible.
- **L'alerte.** Chez GitLab, l'export échouait en silence. Un backup automatique doit prévenir quelqu'un quand il échoue, et vous devez vérifier de temps en temps que le fichier du jour existe et ne pèse pas 0 octet.

4. Complet, différentiel, incrémental

Copier toute la base chaque nuit est simple, mais la copie grossit avec la base. Trois méthodes existent, qui diffèrent par ce qu'elles copient chaque jour.

Trois types de backup, une semaine

Complet

tout, chaque jour
restaurer : 1 fichier



Différentiel

changements depuis dimanche
restaurer : 2 fichiers



Incrémental

changements depuis la veille
restaurer : 7 fichiers



hauteur = taille du backup

D L M M J V S

Prenez une application de 1 Go de données, dont 50 Mo changent chaque jour, avec un backup complet le dimanche.

- Le **backup complet** copie tout, chaque jour. Chaque fichier se suffit à lui-même.
- Le **backup différentiel** copie ce qui a changé **depuis le dernier backup complet**. Le lundi, 50 Mo. Le mardi, 100 Mo. Le samedi, 300 Mo. Pour restaurer, il faut deux fichiers : le complet du dimanche et le différentiel du jour voulu.
- Le **backup incrémental** copie ce qui a changé **depuis le dernier backup, quel qu'il soit**. 50 Mo chaque jour. Pour restaurer le samedi, il faut le complet du dimanche et les six incrémentaux, dans l'ordre. Si l'un d'eux est abîmé, tous les suivants sont inutilisables.

	Complet	Différentiel	Incrémental
Place sur une semaine	7 Go	environ 2 Go	1,3 Go
Durée du backup	Longue	Moyenne, croît dans la semaine	Courte
Restauration	La plus simple	Simple	La plus longue et la plus fragile

Plus le backup est léger à faire, plus la restauration est lourde. Dans la pratique, on combine : un complet par semaine, puis des incrémentaux ou des différentiels entre deux.

`restic` et `borg`, deux outils libres de sauvegarde de fichiers, vont plus loin. Ils découpent les données en blocs et n'envoient que les blocs nouveaux ou modifiés. Chaque backup est incrémental à l'envoi, et se restaure comme un complet.

Deux autres mots reviennent dans les documentations des hébergeurs :

- Un **instantané** (*snapshot*) est une photo d'un disque entier à un moment donné, prise par l'hébergeur. Il se restaure vite, mais il reste chez le même hébergeur, souvent sur la même infrastructure.
- La **restauration à un instant donné** (*point-in-time recovery*, PITR) pousse l'incrémental au bout. La base enregistre en continu le journal de toutes ses écritures. Vous pouvez la remettre dans l'état de 13 h 59, une minute avant le `DROP TABLE`. La perte se compte en secondes.

Pour une base de projet étudiant, qui pèse quelques dizaines de mégaoctets, un export complet chaque nuit suffit. L'incrémental devient utile pour les fichiers envoyés, qui grossissent vite, et le PITR quand chaque minute de données compte.

Ce que coûte la copie hors site

Un stockage d'objets facture la place occupée, au gigaoctet par mois, et parfois la **sortie**, c'est-à-dire les données que vous téléchargez depuis le bucket. Voici les tarifs d'octobre 2026.

Service	Gratuit	Ensuite	Sortie des données
Cloudflare R2	10 Go	0,015 \$ par Go et par mois	Gratuite
Backblaze B2	10 Go	0,007 \$ par Go et par mois	Gratuite jusqu'à trois fois le volume stocké
Amazon S3 (Irlande)	Rien de permanent	0,023 \$ par Go et par mois	100 Go gratuits par mois, puis 0,09 \$ par Go
Hetzner Object Storage	Rien	Forfait de 6,49 € par mois pour 1 To	1 To inclus
Hostkey S3	Rien	Forfait de 4 € par mois pour 250 Go, puis 0,017 € par Go	1 To inclus

La sortie compte le jour de la restauration. Télécharger 500 Go de backups depuis Amazon S3 coûte 36 \$. Depuis R2, rien.

Faites le calcul pour votre projet. Un export de 50 Mo par nuit, gardé 30 jours, occupe 1,5 Go. Ajoutez 2 Go de fichiers envoyés. Le tout tient dans les 10 Go gratuits de R2 ou de B2 : votre copie hors site coûte 0 €.

Le calcul change avec la taille. Pour une base de 50 Go, trente exports complets occupent 1,5 To, soit une vingtaine de dollars par mois chez R2. À cette taille, les backups incrémentaux deviennent rentables, ou un stockage dit « froid », moins cher et prévu pour des archives rarement relues.

Cloudflare demande d'enregistrer un moyen de paiement pour activer R2, même si vous restez dans la partie gratuite. Amazon demande une carte bancaire dès la création du compte. Les sources de tous ces prix sont listées à la fin du chapitre.

5. Automatiser le backup

Un backup lancé à la main est oublié au bout de deux semaines. Quel que soit l'outil, un backup automatique fait toujours les cinq mêmes choses, chaque nuit.

Un backup automatique, cinq étapes

- 1 Exporter**
la base, dans un fichier
- 2 Archiver**
les fichiers envoyés
- 3 Envoyer**
dans un bucket, hors site
- 4 Nettoyer**
les backups trop anciens
- 5 Prévenir**
réussite ou échec, un message

Étape	Ce qui se passe	Outils courants
1. Exporter	La base est lue pendant qu'elle tourne et écrite dans un fichier	<code>pg_dump</code> pour PostgreSQL, <code>mysqldump</code> pour MySQL
2. Archiver	Le dossier des fichiers envoyés devient une archive compressée	<code>tar</code> , ou <code>restic</code> quand le dossier grossit
3. Envoyer	L'export et l'archive partent dans un bucket, hors du serveur	<code>rsync</code> , <code>aws s3</code> , ou l'outil intégré de votre panneau

Étape	Ce qui se passe	Outils courants
4. Nettoyer	Les backups plus vieux que la rétention sont supprimés	Une règle du bucket, ou une commande du script
5. Prévenir	Quelqu'un reçoit un message : réussite ou échec	Un e-mail, un webhook Discord, Healthchecks.io

Trois pièges reviennent, quel que soit l'outil.

- **La version de l'outil d'export.** `pg_dump` doit avoir la même version majeure que la base, ou une version plus récente. Sinon il s'arrête avec une erreur. C'est la panne de GitLab.
- **Synchroniser n'est pas copier.** Une commande comme `rsync sync` rend la destination identique à la source : un fichier supprimé par erreur disparaît aussi de la copie. Pour un backup, on copie (`rsync copy`), on ne synchronise pas.
- **Les fichiers déjà dans un bucket** ne sont pas à l'abri pour autant. Une suppression dans l'application supprime l'objet. Activez le **versionnement** du bucket, qui garde les anciennes versions des objets supprimés ou écrasés, ou copiez-le chaque nuit vers un second.

Être prévenu

C'est l'étape qu'on oublie, et celle qui a manqué à GitLab. Un backup qui échoue en silence est pire qu'une absence de backup : vous croyez être protégés.

Deux mécanismes se complètent. Le script envoie lui-même un message à la fin, par e-mail ou par **webhook**, une adresse fournie par Discord, Slack ou Teams qui affiche dans un salon le texte qu'on lui envoie. Vous voyez chaque matin « Backup : OK », ou « Backup : ÉCHEC ». Mais si le serveur est éteint, aucun message ne part. Un service comme Healthchecks.io fait l'inverse : il attend un signal chaque nuit, et c'est lui qui vous écrit quand le signal manque.

Un exemple de script

Voici à quoi ressemblent les cinq étapes sur un VPS, pour une base PostgreSQL, des fichiers sur le disque et un bucket chez Cloudflare R2.

ATTENTION

Ce script est un exemple, pas une solution à copier. Chaque chemin, chaque nom et chaque outil dépend de votre installation. Adaptez-le, lancez-le à la main, puis restaurez son résultat avant de lui faire confiance.

BASH

```
#!/usr/bin/env bash
# /srv/chat/backup.sh : exemple à adapter
set -euo pipefail # le script s'arrête à la première erreur

source /srv/chat/backup.env # définit DATABASE_URL et DISCORD_WEBHOOK_URL
STAMP=(date+DIR=/var/backups/chatmkdir-p"(date +%F-%H%M) # 2026-10-05-0230
DIR=/var/backups/chat
mkdir -p "(date+DIR=/var/backups/chatmkdir-p"DIR"

# Envoie un message dans un salon Discord
notify() {
    curl -fsS -H "Content-Type: application/json" -d "{\"content\": \"1\"}"
    "DISCORD_WEBHOOK_URL" > /dev/null
}
trap 'notify "Backup du chat : ÉCHEC"' ERR # appelé si une commande échoue

# 1. Exporter la base
pg_dump --format=custom --file="DIR/db-DIR/db-DIR/db-STAMP.dump" "DATABASE_URL"

# 2. Archiver les fichiers envoyés par les utilisateurs
tar -czf "DIR/uploads-STAMP.tar.gz" -C /srv/chat/storage uploads

# 3. Envoyer la copie hors site
rclone copy "DIR" r2:chat-backups

# 4. Nettoyer : 7 jours sur le serveur, 30 jours dans le bucket
find "DIR" -type f -mtime +6 -delete
rclone delete r2:chat-backups --min-age 30d

# 5. Prévenir
notify "Backup du chat : OK (STAMP)"
```

Sur un serveur Linux, c'est **cron**, le planificateur de tâches du système, qui lance ce script chaque nuit. Une ligne suffit, ajoutée avec `sudo crontab -e` :

```
30 2 * * * /srv/chat/backup.sh >> /var/log/backup-chat.log 2>&1
```

Les cinq champs se lisent : minute, heure, jour du mois, mois, jour de la semaine. Vous avez déjà rencontré cette syntaxe dans les workflows GitHub Actions.

6. Quel outil pour votre stack

Écrire son script est rarement nécessaire. Selon votre hébergement, les cinq étapes sont lancées par un outil différent, depuis un endroit différent.

Qui lance le backup ?

VPS nu

cron, sur le serveur

VPS avec Coolify ou Dokploy

le panneau, sur le serveur

PaaS ou Vercel

GitHub Actions, de l'extérieur

Dans tous les cas : une copie dans un bucket.

Pour chaque outil, cette partie dit ce qu'il fait, ce qu'il ne fait pas, et où lire la suite. Les réglages exacts changent d'une version à l'autre : suivez la documentation officielle, pas un tutoriel de 2022.

Les backups de votre fournisseur de VPS

La plupart des hébergeurs de VPS vendent une sauvegarde du disque entier. Chez OVHcloud, chaque VPS reçoit un backup quotidien, gardé 24 heures, sans supplément. L'option Premium garde 7 jours et coûte à partir de 0,30 € HT par mois, selon la taille du disque. Un instantané manuel, utile avant une mise à jour risquée, est au même prix.

C'est le moyen le plus rapide de remettre un serveur entier en route. Mais OVHcloud précise que ces copies sont stockées dans le même centre de données que le VPS. Elles comptent comme une copie, pas comme la copie hors site.

Coolify et Dokploy

Si vous déployez avec un panneau, le backup de la base se règle dans l'interface, sans script.

Dans **Coolify**, l'onglet *Backups* d'une base de données propose une fréquence et une rétention. Coolify lance l'outil d'export dans le conteneur et garde le fichier sur le serveur. Pour la copie hors site, vous déclarez un stockage S3 dans les réglages, puis vous l'activez dans la planification. La restauration se lance depuis *Configuration > Import Backup*.

Pour les fichiers, Coolify archive les volumes d'une application vers le même stockage S3. Un **volume** est le dossier du serveur qu'un conteneur utilise pour garder ses fichiers d'un déploiement à l'autre. Coolify crée ces archives mais n'a pas de bouton pour les restaurer : il faut extraire l'archive à la main.

Dokploy propose la même chose : un onglet *Backups* sur chaque base, avec une destination S3, une fréquence et un bouton *Test*, et des *Volume Backups* pour les fichiers.

Documentation : [backups Coolify](#), [backups Dokploy](#).

Laravel : spatie/laravel-backup

Le paquet `spatie/laravel-backup` est la référence du monde Laravel. Il ajoute trois commandes Artisan, que le planificateur de Laravel lance chaque nuit.

Commande	Ce qu'elle fait
<code>backup:run</code>	Exporte la base, archive les dossiers choisis, envoie un zip vers un disque Laravel, par exemple un bucket S3
<code>backup:clean</code>	Applique la rétention : supprime les zips trop anciens
<code>backup:monitor</code>	Vérifie qu'un backup récent existe, et envoie un e-mail ou un message Slack ou Discord dans le cas contraire

Ce qu'il faut savoir avant de l'installer. Le paquet appelle `pg_dump` ou `mysqldump`, qui doit donc être installé là où tourne l'application. Le planificateur de Laravel doit être lancé par un cron ou par une tâche planifiée du panneau. Les alertes ne partent que si vous indiquez votre adresse et le disque à surveiller dans la configuration. Et le paquet ne restaure pas : vous téléchargez le zip, vous le décompressez, et vous rechargez l'export.

Documentation : spatie.be/docs/laravel-backup.

Django : django-dbbackup

L'équivalent pour Django s'appelle `django-dbbackup`. Il s'appuie sur le système de stockage de Django, donc sur `django-storages` pour écrire dans un bucket S3.

Commande	Ce qu'elle fait
<code>dbbackup</code>	Exporte la base vers le stockage configuré, avec compression et chiffrement en option
<code>mediabackup</code>	Archive le dossier des fichiers envoyés
<code>dbrestore</code> et <code>mediarestore</code>	Rechargent le dernier backup de la base et des fichiers

Django n'a pas de planificateur : les deux premières commandes se lancent depuis un cron du serveur ou une tâche planifiée de votre panneau. Contrairement au paquet Laravel, celui-ci restaure.

Documentation : archmonger.github.io/django-dbbbackup.

Node.js

Des paquets existent sur npm, mais aucun n'a la place de `laravel-backup` dans le monde Laravel. Ce sont de petits projets, peu mis à jour, qui exportent la base puis envoient le fichier dans un bucket. Laravel et Django sont des frameworks complets, avec un planificateur, un système de fichiers et des notifications sur lesquels un paquet de backup peut s'appuyer. Une application Express ou Fastify n'a rien de tout cela en commun avec sa voisine.

Les équipes Node sortent donc le backup de l'application. Elles utilisent le backup du panneau, un script lancé par cron, ou un workflow GitHub Actions. Si vous utilisez Prisma ou Drizzle, rien ne change : ces bibliothèques décrivent le schéma de la base, elles n'exportent pas les données.

Les fichiers : restic et rclone

Deux outils libres reviennent dans toutes les stacks. `rclone` copie des fichiers vers plus de cinquante services de stockage : c'est lui qui envoie une archive dans un bucket, ou qui copie un bucket vers un autre. `restic` fait des backups incrémentaux et chiffrés d'un dossier, directement dans un bucket. Il devient utile quand les fichiers envoyés dépassent quelques gigaoctets.

Documentation : rclone.org, restic.net.

7. Sur une plateforme managée

Sur un PaaS ou en serverless, vous n'avez ni serveur ni cron. La plateforme fait une partie du travail, et cette partie dépend du plan que vous payez. Voici l'état des lieux en octobre 2026.

- **Render.** Sur une base payante, à partir de 6 \$ par mois : restauration à un instant donné sur 3 ou 7 jours selon le plan du compte, et export téléchargeable gardé 7 jours. Sur la base gratuite : aucun backup.
- **Railway.** Backups planifiés des volumes : quotidiens gardés 6 jours, hebdomadaires 27 jours, mensuels 89 jours. Ils demandent le plan Pro, à 20 par mois. Chaque backup est facturé pour la place qu'il occupe, à 0,15 par Go et par mois.
- **Fly.io.** Instantané quotidien de chaque volume, gardé 5 jours, réglable de 1 à 60. Les 10 premiers Go sont gratuits, puis 0,08 par Go et par mois. Le Postgres managé garde ses backups 10 jours et démarre à 38 par mois.
- **Neon.** Restauration à un instant donné sur 6 heures avec le plan gratuit. Sur les plans payants, 1 jour par défaut, réglable jusqu'à 7 ou 30 jours, à 0,20 \$ par Go d'historique et par mois.

- **Supabase.** Plan gratuit : aucun backup. Plan Pro, à 25 *par mois* : *un backup par jour, gardé 7 jours. La restauration à un instant donné est une option à 100* par mois pour 7 jours. Les fichiers du Storage ne sont jamais inclus.
- **Vercel.** Vercel n'héberge pas de base : elle vient de Neon, de Supabase ou d'un autre fournisseur. Vercel Blob coûte à partir de 0,023 \$ par Go et par mois, et n'a pas de backup intégré.

Le chapitre sur l'hébergement comptait les sauvegardes parmi ce que gère un PaaS. C'est vrai sur les plans payants, et chez lui seulement. Toutes les copies sont chez le même fournisseur, attachées au même compte : la règle 3-2-1 n'est pas remplie. Un compte suspendu, une carte refusée ou un projet supprimé par erreur, et les backups partent avec la base. Sur les plans gratuits de Render et de Supabase, aucun backup n'existe.

La copie hors site avec GitHub Actions

Vous n'avez pas de serveur pour lancer un cron, mais vous avez un dépôt GitHub. Un workflow peut se déclencher à heure fixe, avec le déclencheur `schedule` et la même syntaxe que cron. Il fait alors les étapes de la partie 5 depuis une machine de GitHub : il exporte la base avec `pg_dump`, puis dépose le fichier dans un bucket.

Ce workflow n'utilise pas SSH : il n'y a pas de serveur sur lequel se connecter. La machine de GitHub parle directement à deux services, chacun avec son secret, rangé dans les secrets du dépôt comme au chapitre CI/CD.

- **À la base de données**, par le réseau, comme le fait votre application. Le secret est l'adresse de connexion de la base, qui contient l'hôte, l'utilisateur et le mot de passe.
- **Au bucket**, en HTTPS, avec une clé d'API. Vous créez cette clé dans le tableau de bord du stockage, et vous pouvez la limiter à un seul bucket.

Si l'export échoue, le workflow s'arrête et GitHub vous envoie un e-mail : l'alerte est comprise. La rétention se règle du côté du bucket, avec une règle de cycle de vie qui supprime les objets trop anciens.

Deux limites. Sur un VPS, la base n'écoute en général que sur le serveur lui-même, et le workflow ne peut pas l'atteindre : le backup tourne alors sur le serveur. Et dans un dépôt public, GitHub désactive les workflows planifiés après 60 jours sans activité. Sur un projet terminé mais encore en ligne, le backup s'arrête sans prévenir.

Documentation : [le déclencheur `schedule` de GitHub Actions](#).

Les fichiers sur une plateforme managée

Sur Render, Railway ou Fly.io, les fichiers écrits sur un volume sont couverts par les backups ou les instantanés décrits plus haut. Sur Vercel, ils sont dans Vercel Blob ou dans un autre stockage d'objets : activez le versionnement du bucket, ou copiez-le régulièrement vers un second. La documentation de Vercel décrit cette copie périodique.

8. Restaurer : la preuve que le backup existe

Tant que vous n'avez pas restauré un backup, vous ne savez pas s'il fonctionne. Le fichier peut être vide, tronqué, chiffré avec un mot de passe perdu, ou exporté avec une version de l'outil que votre PC ne sait pas lire.

L'essai se fait sur votre PC, sans risque pour la production.



1. **Télécharger** le dernier backup depuis le bucket : l'export de la base et l'archive des fichiers.
2. **Restaurer** l'export dans une base locale vide, et extraire l'archive dans votre projet local.
3. **Vérifier**. Le nombre de lignes doit correspondre à la production, et la date du dernier message à l'heure du backup. Pointez votre application locale sur cette base et ouvrez une conversation : les images s'affichent-elles ?
4. **Noter** le temps total et chaque commande tapée, dans un fichier `RESTORE.md` à la racine du dépôt.

Le temps total, du téléchargement à l'application qui tourne, est la durée réelle de votre arrêt, à comparer à votre RTO. Les commandes notées servent le jour de l'incident, quand personne n'a envie de chercher la bonne option.

Pour un export PostgreSQL fait avec `pg_dump`, les étapes 2 et 3 ressemblent à ceci. Les noms sont à adapter à votre projet.

BASH

```
createdb chat_restore
pg_restore --no-owner --dbname=chat_restore db-2026-10-05-0230.dump
psql chat_restore -c "SELECT count(*), max(created_at) FROM messages;"
```

Un essai réussi une fois ne prouve rien pour la suite. Entre-temps, la base change de version, un dossier de fichiers s'ajoute, une clé d'API expire, et personne ne remarque que le backup ne se restaure plus. Refaites l'essai à date fixe, tous les trois mois par exemple, avec un rappel dans l'agenda du groupe. Refaites-le aussi après chaque changement important : nouvelle version de la base, nouvel hébergeur, nouveau dossier de fichiers.

9. Par où commencer

Votre hébergement	La base	Les fichiers
VPS avec Coolify ou Dokploy	Backup planifié du panneau, ou paquet de votre framework, vers un bucket S3	Backup des volumes du panneau, vers le même bucket
VPS nu	Un script et un cron, ou le paquet de votre framework	Le même script, ou <code>restic</code>
PaaS (Render, Railway, Fly.io)	Les backups de la plateforme s'ils existent sur votre plan, plus un workflow GitHub Actions pour la copie hors site	Les backups de volume de la plateforme
Vercel et base managée	Un workflow GitHub Actions	Versionnement du bucket ou copie vers un second

Dans tous les cas, la liste est la même :

- un backup automatique chaque nuit ;
- une copie dans un bucket qui n'est pas chez votre hébergeur ;
- une rétention d'au moins sept jours ;
- un message en cas d'échec ;
- une restauration essayée, notée, puis refaite tous les trois mois.

Pour votre projet, décrivez une stratégie de backup dans votre rapport et mettez-la en place : elle vous rapporte des points en fonctionnalités supplémentaires. La description dit ce qui est sauvegardé, à quelle fréquence, où vont les copies, combien de temps elles sont gardées, et combien de temps a pris votre essai de restauration.

En résumé

- **Un backup est une copie figée**, que l'application ne peut plus modifier. La réplication, le RAID et la synchronisation n'en sont pas.
- **Quatre choses à protéger** : le code est dans Git, les secrets dans un gestionnaire de mots de passe. Il reste la base et les fichiers envoyés, qui se sauvegardent ensemble.
- **La règle 3-2-1** demande trois copies, deux supports et une copie hors site. Les backups d'un hébergeur restent chez lui : ajoutez un bucket ailleurs.
- **Un backup automatique fait cinq choses** : exporter, archiver, envoyer, nettoyer, prévenir. L'outil change selon la stack, les étapes non.
- **Restaurez un backup, et recommencez tous les trois mois**. Faites l'essai sur votre PC, chronométrez-le et écrivez la procédure.

Que demande la règle 3-2-1 ?

Trois copies des données, sur deux supports différents, dont une hors site, c'est-à-dire ailleurs que chez l'hébergeur de l'application.

Qu'est-ce que le RPO ?

Recovery Point Objective, l'objectif de point de reprise. La perte de données maximale que vous acceptez. Un backup par nuit donne un RPO de 24 heures.

Qu'est-ce que le RTO ?

Recovery Time Objective, l'objectif de délai de reprise. La durée maximale pendant laquelle vous acceptez que le site reste à l'arrêt après un incident.

Que copie un backup différentiel ?

Ce qui a changé depuis le dernier backup complet. Pour restaurer, il faut deux fichiers : le complet et le différentiel du jour voulu.

Que copie un backup incrémental ?

Ce qui a changé depuis le dernier backup, quel qu'il soit. Pour restaurer, il faut le complet et tous les incrémentaux qui le suivent, dans l'ordre.

Pourquoi une base répliquée n'est-elle pas un backup ?

Elle recopie chaque écriture sur le second serveur, y compris la suppression faite par erreur.

Qu'est-ce que la rétention ?

La durée pendant laquelle vous gardez les anciens backups. Trop courte, tous les backups contiennent déjà les données abîmées.

Quelles sont les cinq étapes d'un backup automatique ?

Exporter la base, archiver les fichiers, envoyer la copie hors site, nettoyer les anciens backups, prévenir quelqu'un du résultat.

Pourquoi `rsync` est-il dangereux pour un backup ?

Il rend la destination identique à la source. Un fichier supprimé par erreur disparaît aussi de la copie.

Comment savoir qu'un backup fonctionne ?

En le restaurant, sur son PC, dans une base vide. Tant que ce n'est pas fait, on ne sait pas. L'essai se refait à date fixe, tous les trois mois par exemple.

Widget interactif — à voir en ligne : <https://cours.tsix.be/cours/projet-dintegration-de-developpement/projet-1/stockage-et-backup>

Pour aller plus loin

- Le récit de l'incident GitLab du 31 janvier 2017 : <https://about.gitlab.com/blog/postmortem-of-database-outage-of-january-31/>
- PostgreSQL, `pg_dump` et `pg_restore` : <https://www.postgresql.org/docs/current/backup-dump.html>
- MySQL, `mysqldump` : <https://dev.mysql.com/doc/refman/8.4/en/mysqldump.html>
- Coolify : <https://coolify.io/docs/databases/backups> · Dokploy : <https://docs.dokploy.com/docs/core/databases/backups>
- spatie/laravel-backup : <https://spatie.be/docs/laravel-backup> · django-dbbbackup : <https://archmonger.github.io/django-dbbbackup/>
- rclone : <https://rclone.org> · restic : <https://restic.net> · Healthchecks.io : <https://healthchecks.io>
- Webhooks Discord : <https://support.discord.com/hc/fr/articles/228383668>
- Vercel Blob, exemples de copie de sauvegarde : <https://vercel.com/docs/vercel-blob/examples>

Sources des prix et des durées

Pages officielles consultées le 4 octobre 2026. Les tarifs changent souvent : vérifiez-les avant de choisir.

- Stockage d'objets : [Cloudflare R2](#) · [Backblaze B2](#) · [Amazon S3](#) · [Hetzner Object Storage](#) · [Hostkey S3](#)
- VPS : [options de sauvegarde OVHcloud](#)
- Render : [backups PostgreSQL](#) · [tarifs](#)
- Railway : [backups de volumes](#) · [plans](#)
- Fly.io : [instantanés de volumes](#) · [tarifs](#)
- Neon : [fenêtre d'historique](#) · [tarifs](#)
- Supabase : [backups](#) · [tarifs](#)
- Vercel Blob : [tarifs](#)

CHAPITRE 2

Projet 2 — Application IA-Driven (MVP)

1. Objectif & périmètre

Développer une **application avec l'IA au cœur du produit**, en exploitant les modèles disponibles via **OpenRouter**. L'objectif est de créer un outil qui apporte une **réelle valeur ajoutée**, pas simplement un wrapper d'API basique.

Obligatoire (MVP) :

- Intégration fonctionnelle d'**au moins 1 modèle IA** via OpenRouter
- Fonctionnalité principale exploitant l'IA de manière pertinente
- Authentification + gestion des utilisateurs
- Dashboard administrateur pour **monitoring des coûts IA**
- Landing page + pricing
- Conformité légale de base (RGPD, mentions légales, cookies)

Encouragé (selon pertinence) :

- **Chaînage de prompts** (multi-étapes)
- **Mémoire conversationnelle** (contexte persistant)
- Streaming temps réel (SSE/WebSocket) pour updates d'avancement
- Caching des réponses
- Services tiers (vectorisation, orchestration)

Exemples de catégories : Créativité & Contenu, Éducation, Business & Productivité, Niche verticale, Automation. Voir **Annexe A** pour inspirations détaillées.

2. Livrables & contraintes

- **URL production** publique et fonctionnelle.
- **Repo GitHub** (accès enseignant).
- **Document de suivi hebdomadaire** (à remettre chaque semaine) :
 - Résumé de ce qui a été réalisé
 - Planification de la semaine suivante
 - Screenshots des avancées
 - Répartition : qui a fait quoi
- **Rapport PDF final (cahier des charges)** contenant :
 - **Concept** : problème résolu, public cible

- **Acteurs & use cases**
- **Justification du choix du modèle IA** (coût, performance, pertinence technique)
- **Architecture** (schéma front/back/DB/API IA)
- **Diagramme UML de classes**
- **Dashboard monitoring coûts** (captures d'écran + explication)
- **Stratégie de pricing** basée sur les coûts réels observés (calcul de la marge)
- **Conformité légale** : RGPD, AI Act, mentions légales, gestion cookies (recherches menées par l'équipe)
- **Présentation finale** : captures, parcours utilisateur type
- **CI/CD & déploiement** (pipeline + choix techniques)
- **README** avec instructions de lancement + lien URL prod.
- **Date de remise** : **Dimanche 4 janvier 2026 à 23h59**.

3. Contraintes budgétaires OpenRouter

- **Crédit alloué** : **2\$ par groupe** (tests + développement inclus).
- **Dashboard admin obligatoire** : suivi en temps réel de la consommation (tokens, coûts).
- **Justification du modèle** : expliquer pourquoi ce modèle (GPT-4, Claude, Llama, Mistral...) plutôt qu'un autre (performance/coût/latence).
- **Conseils d'optimisation** (non obligatoires) :
 - Caching des réponses fréquentes
 - Privilégier modèles légers pour tâches simples
 - Prompts concis
 - Chaînage intelligent (petit modèle → gros modèle si besoin)

4. Catégories & exemples d'applications

Vous devez **trouver une idée originale** avec une vraie valeur ajoutée. Voici des catégories pour vous inspirer (voir **Annexe A** pour exemples détaillés) :

Créativité & Contenu

Génération/optimisation de contenu marketing, newsletters, scripts, posts réseaux sociaux...

Éducation & Formation

Tuteurs adaptatifs, correcteurs pédagogiques, simulateurs d'entretien, générateurs d'exercices...

Business & Productivité

Assistants métier, analyse de CVs, veille concurrentielle, génération de documents légaux, feedback clients...

Niche verticale

Santé (avec disclaimers), légal (orientation de premier niveau), RH, fitness, nutrition...

Automation & Workflows

Génération de workflows, gestion de projet assistée, génération de tests, documentation automatique...

Hybride/Original

Personas marketing, analyse de tonalité, brainstorming structuré, simulateurs de débat...

 **À éviter** : chatbot générique sans spécialisation, simple interface pour ChatGPT, traducteur basique, résumeur sans valeur ajoutée.

5. Organisation d'équipe & Git

- Groupes de **2-3 étudiants**.
- **Organisation libre** : répartissez-vous le travail comme vous le souhaitez.
- **Git workflow** :
 - Feature branches : `feature/<nom>`
 - Pull Requests → review → merge
 - **main = production** (déploiement automatique)
- **Minimum 1 PR significative par personne/semaine** (substantielle, testable, documentée).

6. Hébergement & CI/CD

Hébergement

- **Plateformes conseillées** (free tiers) :
 - **Vercel** (Next.js, React, Vue...)
 - **Netlify** (sites statiques, fonctions serverless)
 - **Render** (full-stack, bases de données)
 - **VPS** si vous maîtrisez (DigitalOcean, Hetzner...)
- **Monitoring conseillé** (non obligatoire) : logs, uptime, erreurs.

CI/CD

- **GitHub Actions** (ou GitLab CI, CircleCI...) :
 - Sur **PR** : lint + build (+ tests si présents)
 - Sur **main** : build + **déploiement automatique**
- **Secrets** via variables d'environnement (jamais dans le repo).
- **Suivi** : statut de build visible, logs accessibles.

7. Architecture & technologies

Stacks possibles (exemples)

- **Full JavaScript** : React/Vue (Vite) + Node/Express + OpenRouter SDK
- **Laravel + Vue** (ou Inertia) + OpenRouter API
- **Python** : FastAPI/Django + React + OpenRouter
- **Mobile** : React Native / Flutter + backend au choix
- **Desktop** : Electron / Tauri + frontend web

Vous pouvez proposer d'autres stacks (validation rapide avec l'enseignant).

Streaming temps réel

Encouragé (SSE/WebSocket) pour :

- Réponses IA progressives (token par token)
- Updates d'avancement d'agents IA
- Barres de progression
- Notifications d'étapes intermédiaires

Services tiers

Vous pouvez utiliser des services externes, **mais précisez si payants** :

Vectorisation & embeddings :

- **Supabase Vector** (gratuit avec limitations PostgreSQL)
- **Chroma DB** (open source, self-hosted, gratuit)
- **Weaviate** (open source, cloud gratuit limité)
- **Pinecone** (⚠️ payant, free tier très limité)

Orchestration IA :

- **LangChain** (open source, gratuit)

- **Flowise** (open source, self-hosted)

Autres :

- **Clerk** (auth, free tier)
- **Supabase** (DB + auth + storage, généreux free tier)
- **Firebase** (gratuit avec limites)

8. Fonctionnalités IA minimales (exigées)

- Intégration **OpenRouter** fonctionnelle.
- **Au moins 1 modèle** utilisé de manière pertinente.
- **Prompt engineering** documenté :
 - Prompts optimisés et testés
 - Variables/templates si applicable
 - Gestion du contexte (system prompt, exemples...)
- **Gestion des erreurs** :
 - Timeouts, rate limits API
 - Erreurs de modèle (refus, hallucinations)
 - Fallback ou messages utilisateur clairs
- **Dashboard admin** :
 - Suivi consommation tokens
 - Coûts en temps réel ou quasi-réel
 - Historique des appels (date, modèle, coût)

9. Page marketing & pricing

Landing page

Page publique attractive :

- **Problème résolu** + bénéfices clairs
- **Démonstration** : captures animées, vidéo, GIF
- **CTA évident** : "Essayer maintenant", "Inscription gratuite"...

Pricing

Minimum 2 paliers avec quotas différenciés :

- Exemples : requêtes IA/jour, fonctionnalités premium, rétention historique, export données...
- **Basé sur vos coûts réels** : utilisez le dashboard de monitoring pour calculer votre marge.

- **Transparence** : indiquez que l'application utilise l'IA (modèle, limitations éventuelles).

Exemple :

- **Free** : 10 requêtes/jour, historique 7 jours
- **Pro (9€/mois)** : 500 requêtes/jour, historique illimité, export, support prioritaire

10. Conformité légale & éthique

Vous devrez **mener vos propres recherches** sur la conformité applicable à votre projet :

- **RGPD** :
 - Politique de confidentialité
 - Gestion des données personnelles (collecte, traitement, suppression)
 - Consentement utilisateur si applicable
- **AI Act** (réglementation européenne IA) :
 - Transparence sur l'utilisation de l'IA
 - Disclaimers sur les limitations
 - Classification du risque (si applicable)
- **Mentions légales** complètes (éditeur, hébergeur, contact).
- **Gestion des cookies** : bannière conforme, consentement.
- **Disclaimers spécifiques** si domaine sensible :
 - Santé : "Ne remplace pas un avis médical"
 - Légal : "Ne constitue pas un conseil juridique"
 - Finance : "Ne constitue pas un conseil en investissement"
- **Modération** : si génération de contenu utilisateur, filtres/modération nécessaires.

Ressources : CNIL (RGPD), site officiel AI Act, modèles de mentions légales.

11. Planning indicatif (7 semaines)

Semaine	Objectifs	Livrables
S1	Organisation, choix stack, squelette projet, auth basique, CDC v0	Document suivi S1
S2	Intégration OpenRouter, premiers appels API, déploiement initial (URL prod)	Document suivi S2
S3	CI/CD complet, fonctionnalité IA core MVP, dashboard monitoring coûts	Document suivi S3

Semaine	Objectifs	Livrables
S4	Optimisation prompts, gestion erreurs robuste, caching (si pertinent)	Document suivi S4
S5	Landing page + pricing (basé sur coûts observés), conformité légale	Document suivi S5
S6	Fonctionnalités optionnelles, polish UX, tests utilisateurs	Document suivi S6
S7	Finitions, rapport PDF final, démo fluide en production	Document suivi S7 + Rapport final

12. Suivi hebdomadaire & démo

Document de suivi (chaque semaine)

À remettre **chaque semaine** (format Markdown ou PDF) :

- **Ce qui a été fait** : résumé des avancées
- **Planification S+1** : objectifs de la semaine suivante
- **Screenshots** : preuves visuelles des avancées
- **Répartition** : qui a travaillé sur quoi (commits, PR, tâches)

Démo en direct (revue du jeudi)

1. **URL prod + commit** déployé communiqués.
2. **Scénario live** : démonstration de la fonctionnalité IA principale.
3. **Dashboard coûts** : montrer le monitoring en action.
4. **Bilan** : 1 point fort, 1 point faible, objectif S+1 explicite.

13. Grille d'évaluation

Même grille que le **Projet 1** (critères génériques applicables) :

- [Grille d'évaluation 5IDEV.pdf](#)

Adaptation attendue : qualité de l'intégration IA, pertinence du prompt engineering, monitoring des coûts, conformité légale.

2.1 ANNEXE A — Exemples d'applications & inspirations

CRÉATIVITÉ & CONTENU

Outils existants (inspiration)

- [Jasper.ai](#) — Génération de contenu marketing
- [Copy.ai](#) — Copywriting automatisé
- [Grammarly](#) — Assistant d'écriture avec IA

Idées de projets

- **Générateur de newsletters personnalisées** : agrège RSS + génère digest adapté au profil
 - **Assistant d'écriture de scénarios** : structure en 3 actes, suggestions dialogues
 - **Créateur de posts LinkedIn/Twitter optimisés** : analyse de ton, hooks, prédiction engagement
 - **Générateur de scripts YouTube** : structure intro/corps/CTA, timestamps auto
 - **Outil de refonte multi-plateforme** : article long → thread Twitter + post LinkedIn + story Instagram
-

ÉDUCATION & FORMATION

Outils existants

- [Duolingo Max](#) — Tuteur IA pour langues
- [Khan Academy Khanmigo](#) — Assistant pédagogique
- [Quizlet Q-Chat](#) — Tuteur adaptatif

Idées

- **Correcteur de code pédagogique** : questions socratiques pour guider (pas juste "voilà la solution")
 - **Générateur d'exercices adaptatifs** : analyse niveau → exos progressifs + explications
 - **Simulateur d'entretien d'embauche** : questions adaptées, feedback, scoring
 - **Assistant de révision intelligent** : spaced repetition + quiz depuis notes/PDF
 - **Plateforme de feedback sur essais** : structure, argumentation, style
-



BUSINESS & PRODUCTIVITÉ

Outils existants

- **Notion AI** — Assistant d'écriture intégré
- **Fireflies.ai** — Transcription + résumés réunions
- **Reclaim.ai** — Planification calendrier assistée

Idées

- **Analyseur CVs + générateur lettres motivation** : matching job description → suggestions
 - **Assistant veille concurrentielle** : scrape légal sites → résumé nouveautés
 - **Générateur documents légaux simplifiés** : CGV, contrats, NDA (avec disclaimer)
 - **Outil analyse sentiment client** : agrège reviews → insights + suggestions réponses
 - **Meeting prep assistant** : contexte → agenda suggéré + questions + résumé post
-



NICHE VERTICALE

Outils existants

- **Hippocratic AI** — Assistants santé
- **Harvey AI** — Outil pour avocats (⚠️ payant)
- **Paradox.ai** — Recrutement automatisé

Idées

- **Assistant diététique** : analyse repas (photo/texte) → suggestions nutritionnelles (⚠️ disclaimer médical)
 - **Outil pré-diagnostic symptômes** : chatbot médical avec redirection vers pros (pas de diagnostic)
 - **Générateur plans d'entraînement** : objectifs + niveau → programme adaptatif
 - **Assistant juridique premier niveau** : FAQ droit travail/consommation (disclaimer obligatoire)
 - **Matching freelance/client** : analyse besoins → suggestions profils + pitch
-

AUTOMATION & WORKFLOWS

Outils existants

- **Zapier Central** — Automation avec IA
- **n8n** — Workflows automation open source

Idées

- **Générateur workflows no-code** : description naturelle → schéma d'automatisation
 - **Assistant gestion de projet** : analyse backlog → priorisation, détection blocages
 - **Générateur tests unitaires** : analyse code → génère tests (Jest/PHPUnit)
 - **Dashboard reporting auto** : connecte sources → rapports narratifs hebdo/mensuel
 - **Assistant documentation technique** : analyse codebase → génère/update README, docstrings
-

ORIGINAL/HYBRIDE

Idées innovantes

- **Générateur personas marketing** : inputs entreprise → profils clients détaillés
 - **Analyse tonalité de marque** : scan contenu → suggestions alignement voix
 - **Assistant brainstorming structuré** : SCAMPER, Six Thinking Hats...
 - **Simulateur de débat** : prend position contraire pour tester argumentation
 - **Reformulation multi-registre** : même message → versions formel/casual/technique/marketing
-

Note : Ces exemples sont des **inspirations**, pas des projets imposés. **Originalité et valeur ajoutée attendues** ! 🎯

2.2 🔍 Recherche Textuelle Intelligente

Introduction

Comment Google trouve-t-il les bonnes pages parmi des milliards? Comment un chatbot comprend-il votre question? Ce chapitre explore les **3 techniques fondamentales** qui permettent aux machines de trouver l'information pertinente.

Ce que vous allez découvrir:

1. **TF-IDF** - Mesurer l'importance des mots par leur fréquence

2. **BM25** - L'algorithme standard d'Elasticsearch et Bing

3. **Embeddings** - L'IA qui comprend le sens, pas juste les mots exacts

 **Application interactive:** Testez et comparez les 3 techniques en temps réel →

github.com/opmvpc/recherche

Approche: Du simple (compter des mots) au complexe (IA sémantique). Aucun prérequis nécessaire!