Liberté • Égalité • Fraternité
RÉPUBLIQUE FRANÇAISE

MINISTÈRE CHARGÉ
DE L'EMPLOI

DOSSIER PROFESSIONNEL (DP)

Nom de naissance ▸ *TACHDJIAN*

Nom d'usage ▸ *TACHDJIAN*

Prénom ▸ *Théo*

Adresse ▸ *34bis rue des azalées 13590 Meyreuil*

Titre professionnel visé

Concepteur Développeur d'Application

MODALITÉ D'ACCÈS :

☒ Parcours de formation

DOSSIER PROFESSIONNEL (DP)

☐ Validation des Acquis de l'Expérience (VAE)

DOSSIER PROFESSIONNEL (DP)

Présentation du dossier

Le dossier professionnel (DP) constitue un élément du système de validation du titre professionnel. **Ce titre est délivré par le Ministère chargé de l'emploi.**

Le DP appartient au candidat. Il le conserve, l'actualise durant son parcours et le présente **obligatoirement à chaque session d'examen**.

Pour rédiger le DP, le candidat peut être aidé par un formateur ou par un accompagnateur VAE.

Il est consulté par le jury au moment de la session d'examen.

Pour prendre sa décision, le jury dispose :

1. des résultats de la mise en situation professionnelle complétés, éventuellement, du questionnaire professionnel ou de l'entretien professionnel ou de l'entretien technique ou du questionnement à partir de productions.

2. du **Dossier Professionnel** (DP) dans lequel le candidat a consigné les preuves de sa pratique professionnelle.

3. des résultats des évaluations passées en cours de formation lorsque le candidat évalué est issu d'un parcours de formation

4. de l'entretien final (dans le cadre de la session titre).

[Arrêté du 22 décembre 2015, relatif aux conditions de délivrance des titres professionnels du ministère chargé de l'Emploi]

Ce dossier comporte :

- pour chaque activité-type du titre visé, un à trois exemples de pratique professionnelle ;
- un tableau à renseigner si le candidat souhaite porter à la connaissance du jury la détention d'un titre, d'un

DOSSIER PROFESSIONNEL (DP)

diplôme, d'un certificat de qualification professionnelle (CQP) ou des attestations de formation ;

- une déclaration sur l'honneur à compléter et à signer ;
- des documents illustrant la pratique professionnelle du candidat (facultatif)
- des annexes, si nécessaire.

Pour compléter ce dossier, le candidat dispose d'un site web en accès libre sur le site.

http://travail-emploi.gouv.fr/titres-professionnels

Sommaire

Exemples de pratique professionnelle

DOSSIER PROFESSIONNEL (DP)

EXEMPLES DE PRATIQUE PROFESSIONNELLE

DOSSIER PROFESSIONNEL (DP)

Activité-type 1 Développer une application sécurisée

Exemple n°1 ► Projet CDA : Randomi GO

1. Décrivez les tâches ou opérations que vous avez effectuées, et dans quelles conditions :

Au cours du développement du backend de Randomi GO, j'ai pris en charge l'intégralité du processus d'inscription et d'authentification des utilisateurs, depuis la modélisation de la base jusqu'à la sécurisation de chaque requête. Tout d'abord, j'ai conçu le schéma de la table users dans PostgreSQL en suivant les recommandations Diesel : chaque enregistrement comprend un identifiant unique (UUID), un nom d'utilisateur, une adresse email, un mot de passe haché, un timestamp de création et, le cas échéant, un statut de compte (actif, suspendu). J'ai ensuite implémenté, avec Actix Web, deux endpoints REST :

- **POST /register** qui reçoit un JSON contenant username, email et password. Ce handler valide d'abord la syntaxe des champs (regex pour l'email, longueur minimale du mot de passe), vérifie l'unicité de l'email et du nom d'utilisateur, puis fait appel à la bibliothèque bcrypt pour générer un *hash* sécurisé avec un facteur de coût de 12. Le mot de passe n'est jamais stocké en clair, et la requête échoue proprement si le hachage ou l'insertion SQL se heurte à une violation de contrainte.

- **POST /login** qui prend en entrée username et password. Le service récupère la ligne correspondante en base, compare le mot de passe reçu au *hash* stocké vi bcrypt::verify, et, si tout est correct, génère un JSON Web Token (JWT) signé avec une clé secrète (HS256) et une expiration à 24 heures. Ce token encapsule l'UUID de l'utilisateur, permet le passage du middleware d'authentification sur les routes protégées, et est renvoyé au client dans un cookie HttpOnly pour améliorer la résistance aux attaques XSS.

Sur le plan du frontend, j'ai développé la page login.html entièrement « from scratch » en HTML5, CSS3 et JavaScript Vanilla, sans utiliser de framework externe. Pour garantir l'accessibilité (conformité RGAA 4), j'ai veillé à :

- associer chaque <input> à un <label> explicite, afin que les lecteurs d'écran annoncent

correctement les champs ;

- placer des attributs ARIA (aria-invalid, aria-describedby) pour indiquer les erreurs de validation ;
- utiliser des couleurs contrastées (ratio 4,5:1) et des textes redimensionnables ;
- prévoir des états focus visibles et des messages d'erreur textuels.
  La mise en page repose sur un système de *flexbox* et de *media queries* CSS, garantissant une expérience optimale sur écrans de 320 px (smartphone) jusqu'à 1920 px (desktop). J'ai testé la page avec Lighthouse et avec des émulateurs de navigateurs mobiles, et corrigé les anomalies de responsive design (repli du formulaire sous la charte graphique, boutons tactiles de 44×44 px minimum).

Enfin, pour assurer la robustesse de la logique du jeu, j'ai rédigé un suite complète de tests unitaires en Rust, exécutables via la commande cargo test. Chaque effet de carte – soin de points de vie, attaque, vol d'objets, bonus temporaire – fait l'objet d'au moins trois scénarios : cas nominal, cas limite (zéro cartes, points de vie au maximum/minimum) et comportement en cas d'erreur. Concrètement, j'ai :

1. factorisé le calcul de chaque effet dans une fonction pure (pas d'accès direct à la base), qui prend en entrée l'état du joueur, l'état de l'adversaire et renvoie un *struct* PlayInfo décrivant les modifications (nouveau total de PV, cartes piochées, etc.) ;
2. écrit des tests paramétrés utilisant le macro #[test_case] pour parcourir automatiquement plusieurs combinaisons d'états initiaux ;
3. simulé le RNG (lancer de dé) par injection d'un générateur pseudo-aléatoire déterministe, afin de couvrir à la fois les valeurs minimales, maximales et moyennes sans flakiness ;
4. vérifié la couverture à 100 % des branches critiques avec l'outil argo tarpaulin et corrigé les lignes non couvertes (gestion des erreurs, retours inattendus).

DOSSIER PROFESSIONNEL (DP)

2. Précisez les moyens utilisés :

Pour la partie serveur, j'ai opté pour le langage Rust afin de bénéficier de sa gestion fine de la mémoire et de son système de types rigoureux, et je l'ai associé au framework Actix Web, reconnu pour ses performances élevées et sa prise en charge native des opérations asynchrones. Pour garantir la sécurité des mots de passe en base, j'ai intégré la bibliothèque bcrypt, qui applique un hachage robuste avec facteur de coût paramétrable ; chaque mot de passe utilisateur est transformé en une empreinte infalsifiable avant d'être enregistré. Les données structurées (comptes, relations d'amitié, configuration des salons et historique des parties) sont stockées dans PostgreSQL, une base de données relationnelle éprouvée, à laquelle j'accède grâce à Diesel pour bénéficier d'une génération de schémas et de requêtes SQL typées et optimisées.

Pour valider manuellement chacune des routes exposées par l'API, j'ai utilisé Postman : j'y ai défini des collections de requêtes couvrant l'ensemble des opérations (inscription, authentification, gestion des amis, création et recherche de salons, etc.) et j'ai automatisé l'exécution de ces collections en continu lors des phases d'intégration. Côté client, l'interface est développée en JavaScript "pur" (sans framework externe) : j'y fais appel à l'API Fetch pour envoyer mes requêtes REST et mettre à jour le DOM en fonction des réponses, garantissant ainsi un chargement rapide et un contrôle total sur le code généré. Pour rendre l'application accessible et adaptée à tous les formats d'écran, j'ai mis en place des media queries CSS qui ajustent dynamiquement la disposition des composants — formulaires, listes de joueurs, plateau de jeu — aussi bien sur mobile que sur grand écran.

Pour prototyper les maquettes et tester les parcours utilisateurs avant même d'écrire une ligne de code, je me suis appuyé sur Figma, ce qui m'a permis de collaborer en temps réel avec mes coéquipiers et d'ajuster rapidement les retours visuels. Une fois les premières pages développées, j'ai lancé des audits via Lighthouse pour évaluer l'accessibilité, la performance et les bonnes pratiques : cet outil m'a guidé dans l'amélioration des contrastes, de la structure des balises ARIA et de la rapidité de chargement.

Enfin, sur le volet des tests automatisés, j'ai utilisé la macro assert_eq! de Rust pour écrire des tests unitaires simples et lisibles, complétés par des tests paramétrés qui parcourent automatiquement plusieurs scénarios d'entrée. Pour simuler l'état d'un joueur ou les réponses de l'API dans ces tests, j'ai tiré parti de serde_json pour créer facilement des objets JSON factices, ce qui m'a permis de valider la logique métier sans avoir à instancier une base de données complète. Pour mesurer la qualité de cette couverture de tests, j'ai intégré l'outil tarpaulin qui génère des rapports détaillés : je pouvais ainsi identifier les branches non couvertes et ajouter juste ce qu'il fallait de cas de test pour atteindre un

taux de couverture optimal avant chaque déploiement.

3. Avec qui avez-vous travaillé ?

Erwan Blancard Driss Khelfi

4. Contexte

Nom de l'entreprise, organisme ou association ► *La Plateforme_*

Chantier, atelier, service ► *Dans le cadre de la formation : Bachelor IT Développeur Logiciel*

Période d'exercice ► Du : *06/01/2025* au : *31/07/2025*

5. Informations complémentaires *(facultatif)*

J'ai d'abord conçu une couche de filtrage et de validation systématique de toutes les données reçues côté serveur, en m'assurant que chaque champ soumis respecte des règles strictes (format, longueur maximale, absence de caractères interdits) et en neutralisant toute tentative de code malicieux avant qu'elle n'atteigne la base de données. En cas de données invalides ou suspectes, l'API renvoie toujours une réponse structurée sous forme JSON, explicitant le champ concerné, le type d'erreur et un message clair à destination du client, ce qui facilite le débogage et l'affichage d'un retour utilisateur cohérent sans divulguer d'informations sensibles du système.

Sur la partie interface, j'ai mis en place une grille fluide associée à des points de rupture soignés qui garantissent une présentation irréprochable sur tous les écrans, du plus petit smartphone à 320 pixels

jusqu'aux écrans 4K à 1920 pixels et au-delà. Chaque élément — boutons, formulaires, zones de texte et têtes de section — s'adapte de façon harmonieuse, sans recourir à des ajustements artisanaux, et j'ai veillé à respecter un contraste minimal de 4,5 pour un, comme préconisé par les normes WCAG niveau AA, afin d'assurer une lisibilité optimale pour tous les utilisateurs.

Pour que ces garanties de sécurité, de robustesse et d'accessibilité soient maintenues en continu, j'ai développé un ensemble de tests automatisés—tests unitaires pour la validation des schémas d'entrée, tests d'intégration simulant des soumissions malveillantes, et audits d'accessibilité ciblant le contraste et la réactivité. Ces suites de vérifications sont déclenchées automatiquement dans ma chaîne d'intégration continue sur GitHub Actions, de sorte qu'à chaque mise à jour du code ou à chaque nouvelle branche, l'ensemble des contrôles se lance sans intervention humaine, et tout éventuel échec bloque la progression jusqu'à résolution des anomalies.

DOSSIER PROFESSIONNEL (DP)

Activité-type 2

Concevoir et développer une application sécurisée organisée en couches

Exemple n° 1 ► Projet CDA : Randomi GO

1. Décrivez les tâches ou opérations que vous avez effectuées, et dans quelles conditions :

À partir des besoins fonctionnels du jeu Randomi GO — c'est-à-dire la gestion des comptes utilisateurs, la constitution de listes d'amis, la création et la participation à des salons de jeu, ainsi que le suivi des sessions de parties en cours — j'ai entamé la phase de conception par une modélisation MERISE exhaustive. J'ai d'abord identifié les entités clés : **Account** pour représenter chaque utilisateur inscrit, **FriendRequest** et **Friendship** pour décrire respectivement l'envoi d'une demande d'ami et la concrétisation du lien d'amitié, **Lobby** pour regrouper les paramètres d'une salle de jeu (nombre maximal de joueurs, règles spécifiques, identifiant unique) et **GameSession** pour tracer l'état d'une partie en cours (ordre de jeu, historique des actions, scores détaillés).

J'ai ensuite précisé les associations et leurs cardinalités : un compte peut envoyer et recevoir plusieurs demandes d'amis, chaque salon peut ouvrir ou fermer l'accès à plusieurs comptes, et chaque session de jeu reste liée à un seul salon. À partir de ce schéma conceptuel, j'ai normalisé les données jusqu'à la troisième forme normale, puis traduit le tout en schéma relationnel pour PostgreSQL. Grâce aux migrations Diesel, j'ai créé les tables suivantes :

- **accounts** : identifiant de type UUID en clé primaire, nom d'utilisateur, adresse email, mot de passe haché, date de création et statut du compte,
- **friend_requests** : identifiant UUID PK, identifiant de l'émetteur et identifiant du destinataire — tous deux références vers accounts.id —, statut de la demande et date d'envoi,
- **friendships** : identifiant UUID PK, paire de comptes liés (account_a_id et account_b_id) et date de création de l'amitié,
- **lobbies** : identifiant UUID PK, identifiant du créateur (owner_id → accounts.id), nom de la salle, nombre maximum de joueurs, champ parameters enregistré en JSONB et date de création,

- **game_sessions** : identifiant UUID PK, référence vers la salle (lobby_id → lobbies.id), état de la partie stocké en JSONB (cartes en main, scores, tour en cours), date de début et date de fin.

Pour l'architecture applicative, j'ai opté pour un modèle en trois couches afin de garantir la séparation des responsabilités et de faciliter la montée en charge :

1. **Routeur et contrôleurs (couche Web/API)**
   J'ai choisi Actix Web comme cœur du service HTTP. Le routeur principal est configuré pour prendre en charge les chemins slash auth, slash accounts, slash friends, slash lobbies et slash game. Chaque contrôleur regroupe un ensemble de routes dédié : par exemple, AuthController couvre les requêtes POST slash register et POST slash login, tandis que LobbyController gère POST slash lobbies slash create et GET slash lobbies slash :id. Les middlewares successifs assurent le CORS, la vérification du jeton JWT et la journalisation des appels. Les extracteurs Actix web point Json angle bracket T angle bracket, web point Path angle bracket Uuid angle bracket et web point Data angle bracket DbPool angle bracket permettent respectivement de désérialiser automatiquement le corps JSON, de récupérer les paramètres d'URL typés et d'injecter la connexion à la base de données. Cette couche renvoie des réponses uniformes au format JSON, assorties des codes HTTP adéquats (200, 201, 400, 401, 404, 500).

2. **Service de Jeu (couche WebSocket et logique temps réel)**
   Pour la communication bi-directionnelle entre les clients d'une même partie, j'ai mis en place un module GameService basé sur les WebSockets d'Actix. Lorsqu'une partie démarre, chaque joueur se connecte à l'URL slash game slash ws avec son jeton comme paramètre (token égal JWT). Le service crée un "hub" isolé par salon, géré par un acteur qui conserve l'état du jeu (cartes en main, scores, tour en cours). Les messages entrants — typés comme PlayCard avec player_id et card_id, ou EndTurn avec player_id — sont routés vers l'acteur concerné. Celui-ci applique la logique métier : désérialisation, vérification des règles, mise à jour de l'état, puis diffuse aux clients un message GameUpdate contenant le nouvel état. Ce découplage par acteurs garantit qu'une partie reste indépendante des autres et permet d'équilibrer la charge sur plusieurs instances si besoin.

3. **Accès aux données (couche persistance Diesel + PoloDB)**
   Pour les données strictement relationnelles (comptes, amis, historique des sessions), j'ai adopté Diesel comme ORM, ce qui m'a permis de générer automatiquement les schémas Rust à partir des migrations SQL et d'écrire des requêtes fortement typées, sécurisées et optimisées. Toutefois, la flexibilité requise pour stocker les paramètres dynamiques des salons — règles de jeu évolutives, configurations JSON imbriquées, historique de chat — rendait un schéma

purement relationnel trop rigide. J'ai donc introduit PoloDB, une base NoSQL embarquée en Rust parfaitement intégrée avec Serde pour la (dé)sérialisation. Chaque salon est sauvegardé dans PoloDB sous forme de document JSON sans schéma fixe : paramètres initiaux (max_players, mode de jeu, etc.) et historique d'événements (journal de chat, draft de cartes) coexistent dans un unique objet persistant en mémoire et sur disque. Cette solution offre à la fois la rapidité d'accès nécessaire pour afficher et mettre à jour les lobbies en temps réel, et la souplesse d'y ajouter de nouveaux champs de configuration sans recourir à des migrations SQL.

En combinant la robustesse de PostgreSQL pour les données relationnelles critiques et la souplesse de PoloDB pour les entités à structure variable, l'application Randomi GO bénéficie à la fois de la sécurité ACID et de la liberté offerte par un modèle document. Cette architecture en couches, assortie d'acteurs isolés pour le temps réel, garantit un code organisé, maintenable et facilement extensible.

2. Précisez les moyens utilisés :

Au cours de la conception et du développement de la plateforme Randomi GO, j'ai utilisé DBeaver comme environnement de modélisation graphique pour élaborer et visualiser l'intégralité du schéma de base de données. Grâce à sa capacité de reverse-engineering, j'ai pu tester plusieurs versions du modèle relationnel, ajuster finement les relations entre les entités (comptes, amis, salons, sessions de jeu) et générer automatiquement les scripts SQL compatibles avec PostgreSQL. Cette phase exploratoire m'a non seulement permis de valider la cohérence de mes choix de conception, mais aussi de produire un diagramme entité-relation complet, exporté en annexe du dossier, qui sert de référence visuelle pour toute l'équipe.

Pour versionner et appliquer ces changements de schéma, j'ai mis en place les migrations Diesel. À chaque modification du modèle, j'ai exécuté la commande « diesel migration generate » pour créer un couple de fichiers de migration contenant les instructions de création ou de mise à jour des tables, ainsi que leur équivalent de rollback. Ces scripts, stockés dans le dépôt, garantissent que chaque instance de la base qu'il s'agisse de l'environnement de développement, de test ou de production reste strictement synchronisée avec le code de l'API.

Sur le plan applicatif, j'ai structuré le back-end selon le patron Modèle-Vue-Contrôleur. Les contrôleurs reçoivent les requêtes HTTP, invoquent les services métier pour la logique de l'application, puis renvoient les données formatées en JSON vers le client. Cette séparation en trois couches facilite la

maintenance du code, l'écriture de tests unitaires ciblés et l'évolution des fonctionnalités sans entraîner d'effets de bord.

Pour la gestion du temps réel, j'ai intégré une couche WebSocket directement dans le service de jeu. Dès qu'une partie démarre, chaque joueur établit une connexion permanente à une URL dédiée, qui bascule en WebSocket. J'ai utilisé les capacités natives de mon framework pour accepter la montée en socket, puis j'ai mis en place un système d'acteurs un « hub » par salon qui gère l'état du jeu (cartes en main, scores, tour en cours). Chaque action d'un joueur est envoyée sous forme de message typé, traitée par l'acteur correspondant, puis redistribuée en temps réel à tous les participants, assurant une latence minimale et un échange bidirectionnel continu.

Pour stocker les données de salon, qui peuvent évoluer librement (règles de partie personnalisées, historique de discussion, paramètres dynamiques), j'ai fait le choix d'intégrer le crate PoloDB. Cette base NoSQL embarquée fonctionne entièrement en mémoire, avec persistance sur disque, et ne nécessite pas de schéma fixe. J'y sauvegarde chaque lobby sous forme de document JSON, me donnant la liberté d'ajouter ou de modifier des champs sans toucher aux migrations relationnelles.

La sérialisation et la désérialisation de ces documents, ainsi que de tous les échanges JSON entre le client et l'API, sont assurées par le crate Serde. Grâce à ses macros, chaque structure Rust annotée devient automatiquement convertible vers et depuis JSON, ce qui élimine toute logique de parsing manuelle et garantit une correspondance parfaite entre les types Rust et le format transporté.

Enfin, pour vérifier la tenue de l'ensemble sous forte charge, j'ai conduit des tests de montée en charge en local à l'aide de l'outil k6. J'ai simulé plusieurs centaines de requêtes HTTP simultanées et des dizaines de connexions WebSocket persistantes, tout en mesurant les temps de réponse, l'utilisation CPU et mémoire. Ces tests m'ont permis d'ajuster la taille du pool de connexions à la base de données, d'optimiser la configuration des workers du serveur et de mettre en place des limites de ressources Docker, assurant ainsi une plateforme stable et réactive même en cas de pics d'activité.

3. Avec qui avez-vous travaillé ?

Erwan Blancard Driss Khelfi

4. Contexte

Nom de l'entreprise, organisme ou association ► *La Plateforme_*

Chantier, atelier, service ► *Dans le cadre de la formation : Bachelor IT Développeur Logiciel*

Période d'exercice ► Du : *06/01/2025* au : *31/07/2025*

5. Informations complémentaires *(facultatif)*

Au niveau de la base de données, j'ai veillé à ce que chaque colonne porte un nom explicite en majuscules avec des traits de soulignement pour séparer les mots, conformément à la convention SCREAMING_SNAKE_CASE. Par exemple, les identifiants et les dates sont nommés ID, CREATED_AT ou UPDATED_AT, tandis que les champs de liaison tels que USER_ID ou LOBBY_ID utilisent la même logique afin d'assurer une parfaite homogénéité dans l'ensemble du schéma. Cette stratégie de nommage présente plusieurs avantages : elle facilite la lecture et l'écriture des requêtes SQL, elle garantit l'absence de collisions de noms dans les jointures, et elle se prête particulièrement bien aux générateurs de code et aux outils d'ORM qui s'appuient sur ces conventions pour créer automatiquement les structures de données dans le code.

De plus, afin d'optimiser les performances des opérations de jointure et de recherche, j'ai créé des index systématiques sur chacune des colonnes faisant office de clé étrangère. Ainsi, dès qu'une requête cherche à combiner les tables users et friend_requests, ou à récupérer toutes les parties associées à un salon précis, PostgreSQL peut utiliser ces index pour localiser très rapidement les lignes concernées, au lieu d'effectuer un balayage complet de la table. J'ai également ajouté des index composites sur les colonnes fréquemment interrogées ensemble, comme LOBBY_ID et CREATED_AT dans la table game_sessions, ce qui permet d'accélérer les tris chronologiques et les recherches par session dans un même passage.

Pour illustrer le fonctionnement en temps réel de Randomi GO, j'ai produit un diagramme de séquence détaillé qui documente pas à pas l'échange entre le client, le module de gestion des salons de jeu, le cœur du service de jeu et enfin la diffusion aux différents clients connectés. Dans ce diagramme, on voit

d'abord le client envoyer une requête de création ou de jointure de salon, puis le service de lobby vérifier les droits et renvoyer un accusé de réception. Ensuite le client initie la connexion WebSocket au service de partie, qui instancie un nouvel acteur de jeu et lui fournit l'état initial. À chaque action de joueur par exemple poser une carte ou terminer son tour un message est acheminé vers cet acteur, la logique métier est appliquée pour mettre à jour l'état, et une réponse structurée est renvoyée aux autres participants pour rafraîchir leur interface. Ce schéma de communication, exporté en annexe sous forme de graphique UML, sert à la fois de guide pour le développement et de documentation pour toute extension future du protocole de messages.

DOSSIER PROFESSIONNEL (DP)

Activité-type 3

Préparer le déploiement d'une application sécurisée

Exemple n° 1 ► Projet CDA : Randomi GO

1. Décrivez les tâches ou opérations que vous avez effectuées, et dans quelles conditions :

J'ai mis en place un pipeline CI/CD basé sur GitHub Actions, défini dans un fichier unique stocké à la racine du dépôt. Ce workflow est déclenché à chaque « push » sur la branche principale, mais aussi sur les branches de fonctionnalités, afin de garantir la qualité et la stabilité du code avant toute fusion. Son exécution se répartit en plusieurs étapes successives :

Tout d'abord, une étape de linting vérifie le style et la qualité du code Rust à l'aide de l'outil Clippy, puis analyse les fichiers JavaScript et TypeScript avec ESLint. Cette double passe permet de détecter les erreurs de syntaxe, les mauvaises pratiques et les incohérences stylistiques avant même la compilation.

Ensuite, le code Rust est compilé en mode « debug » pour s'assurer que tous les modules se lient correctement, puis en mode « release » pour valider l'absence de warnings et préparer la construction d'image optimisée.

Une fois la compilation validée, les tests unitaires Rust sont lancés. Chaque fonction critique est couverte et génère un rapport de couverture. En parallèle, un job distinct exécute les tests d'intégration Postman via Newman, en important la collection JSON et l'environnement depuis le dépôt. Ceci garantit que toutes les routes de l'API fonctionnent bien avec une vraie base de test.

Vient ensuite une phase de tests end-to-end Playwright : un navigateur sans interface graphique se lance pour parcourir automatiquement les pages clés du front, remplir les formulaires d'inscription et de connexion, créer et rejoindre un salon, et simuler quelques tours de jeu. Les éventuelles différences visuelles ou plantages y sont immédiatement détectés.

Lorsque tous ces contrôles sont passés avec succès, le workflow construit les images Docker pour chaque composant : l'API Rust, le frontend Vite, et un conteneur de tests si nécessaire. Chacune est taguée à la fois avec le numéro de commit et avec « latest », puis poussée sur Docker Hub en utilisant des identifiants stockés en secrets GitHub, ce qui permet un déploiement instantané sur n'importe quel environnement.

Parallèlement, j'ai rédigé un fichier docker-compose.yml pour orchestrer localement l'ensemble des services de Randomi GO en développement. Ce document décrit notamment :

Un service de base de données sous Postgres : je spécifie l'image officielle postgres :14-alpine, j'ai défini les variables d'environnement POSTGRES_USER, POSTGRES_PASSWORD et POSTGRES_DB pour initialiser la base. J'attache deux volumes Docker : l'un pour stocker les données de façon persistante, l'autre pour monter le dossier migrations, de sorte qu'au démarrage chaque script SQL placé dans migrations/**/up.sql soit automatiquement exécuté par un petit script shell embarqué.

Un service API hébergé sur Actix Web, qui dépend explicitement de la base de données. Le conteneur est construit à partir du Dockerfile du projet Rust, expose le port 8080 en TCP et reçoit en variables d'environnement l'URL de connexion à Postgres ainsi que le chemin vers le dossier de stockage de PoloDB. J'ai également fixé une limite de ressources (mémoire et CPU) pour éviter toute saturation lors de tests simultanés.

Un service frontend composé de deux phases : d'abord, un conteneur Node.js installe les dépendances et lance la compilation Vite pour générer le dossier dist ; ensuite, ces fichiers statiques sont servis par un conteneur Nginx minimal. Le fichier de configuration Nginx a été ajusté pour rediriger toutes les requêtes vers index.html en mode « history » et pour compresser automatiquement les fichiers CSS et JavaScript. Le port 80 est exposé, et un volume « cache » permet de réutiliser les modules Node entre deux builds pour accélérer les itérations.

Grâce à ce docker-compose, il suffit d'exécuter « docker-compose up --build » pour retrouver en local l'intégralité de la plateforme : base de données initialisée, API prête à recevoir des requêtes, et frontend interactif. Cette configuration reflète fidèlement l'environnement de production, tout en restant légère et rapide à déployer pour le développement et les tests continus.

2. Précisez les moyens utilisés :

J'ai commencé par rédiger un unique fichier de configuration pour GitHub Actions, écrit au format YAML, qui décrit l'ensemble du pipeline de livraison continue. Ce document définit plusieurs « jobs » ordonnés : l'installation des dépendances, le contrôle de qualité du code, la compilation, l'exécution des tests unitaires et d'intégration, la construction des images Docker puis leur publication vers un registre distant. Chaque étape est déclenchée automatiquement lors de chaque « push » ou création de « pull request », garantissant ainsi qu'aucune modification n'échappe au processus de validation.

Pour sécuriser l'accès au registre Docker Hub et aux serveurs de déploiement, j'ai stocké dans GitHub des secrets dédiés : les identifiants du compte Docker Hub ainsi que les clés SSH nécessaires pour se connecter aux machines de production. Ces valeurs restent invisibles dans les journaux d'exécution et ne sont injectées dans l'environnement qu'au moment opportun, évitant toute fuite d'information sensible.

Côté conteneurisation, j'ai utilisé Docker pour construire les différentes images — base de données, API et frontend — puis j'ai orchestré leur démarrage local avec Docker Compose. Le fichier de composition contient la description de chaque service, leurs liens réseaux, les ports exposés et le montage des volumes. Cette approche me permet de lancer l'ensemble de la plateforme en une seule commande, quel que soit le poste de travail ou l'environnement.

Pour garantir que la base de données PostgreSQL soit toujours à jour, j'ai écrit un petit script d'entrée qui, au démarrage du conteneur, parcourt le dossier des migrations SQL et les applique automatiquement dans l'ordre chronologique. Ce mécanisme évite tout décalage entre le schéma attendu par l'application et celui réellement présent en base, simplifiant la mise en place de nouveaux environnements.

Toutes les options de configuration — chaînes de connexion, clés secrètes, ports, chemins d'accès — sont centralisées dans un seul fichier .env situé à la racine du projet. Docker Compose le charge automatiquement, et l'application elle-même lit ces variables dès son initialisation. Cette organisation rend la maintenance plus simple et évite de disséminer des paramètres sensibles dans plusieurs scripts ou documents.

Pour tester le service dans des conditions proches de la production, j'ai mis en place un environnement de test complet basé sur Docker Compose. Il inclut une instance de PostgreSQL, l'API compilée en mode test et un serveur Nginx simulant la mise en ligne du frontend. Grâce à ce banc de tests, je peux lancer

automatiquement toutes les suites d'intégration et vérifier que le système se comporte correctement avant de valider une nouvelle version.

Enfin, afin de faciliter l'analyse en cas de problème, j'ai configuré la génération d'un rapport au format HTML à la fin de chaque exécution de tests end-to-end. Ce rapport intègre des captures d'écran des pages au moment où une assertion a échoué, ainsi que les journaux d'erreur détaillés. Il est archivé dans les artefacts du workflow GitHub Actions, ce qui permet de l'examiner facilement sans relancer les tests manuellement.

3. Avec qui avez-vous travaillé ?

Erwan Blancard Driss Khelfi

4. Contexte

Nom de l'entreprise, organisme ou association ► *La Plateforme_*

Chantier, atelier, service ► *Dans le cadre de la formation : Bachelor IT Développeur Logiciel*

Période d'exercice ► Du : *06/01/2025* au : *31/07/2025*

5. Informations complémentaires *(facultatif)*

Pour garantir l'intégrité du code livré, j'ai configuré la branche principale de notre dépôt comme une branche protégée : aucun contributeur ne peut fusionner une nouvelle fonctionnalité tant que

l'ensemble des vérifications automatisées n'a pas été validé avec succès. Concrètement, cela signifie que chaque fois qu'une pull request est ouverte, GitHub vérifie que tous les jobs du pipeline — linting, compilation, tests unitaires, tests d'intégration, contrôles end-to-end et construction d'images Docker — sont bien passés avant d'autoriser la fusion sur la branche principale. Ce mécanisme évite qu'un code défectueux ou inachevé puisse contaminer l'environnement de production.

Une fois que la branche principale a été mise à jour, le déploiement est orchestré depuis un serveur de build dédié. J'ai rédigé un script d'automatisation qui se connecte par SSH aux machines hébergées sur la plateforme DigitalOcean, en utilisant une clé privée sécurisée stockée dans nos secrets GitHub. Le script déclenche une mise à jour progressive de chaque instance — ce que l'on appelle un déploiement par rolling update — en relevant un conteneur à jour sur une première machine, en vérifiant son état de santé, puis en passant à la suivante. Cette approche garantit qu'au moins une instance reste toujours disponible pour servir les utilisateurs, assurant une continuité de service sans interruption perceptible.

Tous les tests automatisés sont exécutés en amont dans le pipeline et sont configurés de manière à bloquer toute étape de publication ou de déploiement si une seule assertion échoue. Si une régression est détectée — par exemple un test unitaire qui ne couvre plus le calcul des points ou une vérification d'intégration qui ne parvient plus à joindre la base de données — le job concerné passe en échec, la pull request reste ouverte et la mise en production est suspendue jusqu'à correction du problème. Ce fonctionnement "stop on failure" nous permet de détecter immédiatement toute anomalie introduite, de corriger le tir avant que le code ne touche la branche principale et, a fortiori, l'environnement de production.

DOSSIER PROFESSIONNEL (DP)

Titres, diplômes, CQP, attestations de formation

(facultatif)

Intitulé	Autorité ou organisme	Date
Baccalauréat Scientifique	**Lycée Marie Madeleine Fourcade**	2020

Déclaration sur l'honneur

Je soussigné(e) [prénom et nom] Théo TACHDJIAN ,

déclare sur l'honneur que les renseignements fournis dans ce dossier sont exacts et que je suis l'auteur(e) des réalisations jointes.

Fait à Meyreuil le 31/07/2025

pour faire valoir ce que de droit.

Signature :

DOSSIER PROFESSIONNEL (DP)

Documents illustrant la pratique professionnelle

(facultatif)

Intitulé
Cliquez ici pour taper du texte.

DOSSIER PROFESSIONNEL (DP)

ANNEXES

(Si le RC le prévoit)