Ouvert le : mercredi, 27 août 2025, 00:00
À rendre : dimanche, 28 septembre 2025, 23:59

Contexte

Dans le cadre de ce projet d'architecture logicielle avancée, vous assumez le rôle d'**architecte logiciel** mandaté par l'entreprise fictive **BrokerX+**, une plateforme émergente de courtage en ligne. L'organisation souhaite moderniser son système de gestion des opérations de courtage, actuellement limité, afin de répondre à la croissance de sa clientèle et aux exigences de fiabilité, de sécurité et de performance définies dans le cahier de charge du projet BrokerX.

Votre mandat consiste à poser les **fondations architecturales** du système. Pour cette première phase, vous devez :

- Concevoir une **architecture monolithique initiale** claire, cohérente et évolutive, en vous inspirant des bonnes pratiques de conception étudiées en cours.
- Définir les **cas d'utilisation prioritaires** (Must-Have) et les modéliser sous forme de scénarios.
- Documenter l'architecture selon une approche **4+1 views** et compléter au moins deux **Architectural Decision Records (ADR)** justifiant vos choix structurants.
- Proposer une stratégie de **gestion des exigences de qualité** (performance de base, disponibilité, sécurité, observabilité minimale par journalisation).
- Démontrer la faisabilité technique par une première implémentation (prototype monolithique conteneurisé) et un pipeline **CI/CD** rudimentaire.

En tant qu'architecte logiciel, vous êtes responsable de :

- **Analyser le cahier de charge** et traduire les besoins métiers en exigences architecturales.
- **Prendre des décisions structurantes** (patterns, couches logicielles, choix technologiques) et les justifier.
- **Communiquer clairement** l'architecture au reste de l'équipe par des modèles, des vues et des documents de référence.
- **Préparer l'évolution** du système vers des architectures distribuées (API RESTful, microservices, événements), qui seront abordées dans les phases ultérieures.

Objectifs d'apprentissage

À l'issue de cette première phase, l'étudiant·e devra être capable de :

1. **Concevoir une architecture monolithique évolutive**
    - Déployer un système initial fonctionnel dans la **machine virtuelle fournie**
2. **Appliquer les principes du Domain-Driven Design (DDD)**
    - Identifier les **bounded contexts**, définir un **modèle de domaine clair**, et assurer une séparation logique des responsabilités.
3. **Mettre en œuvre des patrons de conception adaptés**
    - Intégrer et justifier l'usage de patrons tels que **MVC**, **Architecture hexagonale** ou **GoF**, afin d'assurer la modularité, la maintenabilité et l'évolutivité du système.
4. **Concevoir une solution de persistance robuste et fiable**
    - Implémenter une couche de persistance cohérente (ORM ou DAO), garantissant l'intégrité des données et leur accessibilité dans le prototype.
5. **Documenter et justifier les choix architecturaux**
    - Produire des **diagrammes UML** (4+1 views), compléter des **Architectural Decision Records (ADR)**, et structurer la documentation selon le modèle **Arc42** pour assurer la traçabilité et la communication des décisions.
6. **Implémenter un prototype réaliste et fonctionnel**
    - Développer une première version exécutable couvrant les **cas d'utilisation prioritaires**, démontrant la faisabilité technique et le respect des exigences minimales.
7. **Assurer la qualité par les tests automatisés**
    - Établir une suite de tests **unitaires**, **d'intégration** et **end-to-end**, afin de valider la robustesse du prototype et de détecter les régressions.
8. **Mettre en place et enrichir les pratiques DevOps**
    - Maintenir et étendre les pratiques de **CI/CD** et de **conteneurisation** (Docker), en intégrant les étapes de compilation, tests e déploiement automatisé.

?

Tâches à réaliser (livrables et Critère d'acceptation)

1) Analyse métier & DDD

1. **Clarifier le périmètre initial & cas d'utilisation Must**
   - Livrables : UC textuels (scénarios principal/alternatifs), priorisation MoSCoW.
   - CA : au moins 3 UC Must entièrement décrits et validés.
2. **Cartographier le domaine (DDD)**
   - Livrables : **bounded contexts**, **ubiquitous language**, diagramme de contexte, esquisse du **modèle de domaine**.
   - CA : glossaire validé, entités/agrégats identifiés pour les UC Must.

2) Architecture & décisions

1. **Concevoir l'architecture monolithique évolutive**
   - Livrables : choix de style (**Hexagonal** ou **MVC** + ports/adapters), couches & dépendances.
   - CA : dépendances dirigées (pas de cycles), couplage contrôlé aux frameworks.
2. **Documenter l'architecture (4+1 + Arc42)**
   - Livrables :
     - 4+1 : Vues **Logique**, **Processus (C&C)**, **Déploiement**, **Développement**, **Scénarios**.
     - **Arc42** : sections 1–8 (contexte, contraintes, concept solution, décisions, qualité, risques).
   - CA : chaque vue contient diagramme + texte (contexte/éléments/relations/rationnel).
3. **Consigner les décisions structurantes (ADR)**
   - Livrables : ≥ **3 ADR** (architecture hexagonale vs MVC, persistance, stratégies d'erreurs/versionnage interne).
   - CA : format ADR complet (statut, contexte, décision, conséquences).

3) Persistance & intégrité

1. **Concevoir le schéma de persistance**
   - Livrables : modèle logique (ER/UML), choix **ORM ou DAO**, transactions, contraintes d'intégrité.
   - CA : migrations reproductibles; données seed pour démo.
2. **Implémenter la couche de persistance**
   - Livrables : repositories/DAOs, mapping, tests d'intégration sur DB conteneurisée.
   - CA : CRUD robuste sur agrégats clés; rollback sur erreurs; **idempotency key** si nécessaire.

4) Implémentation du prototype

1. **Implémenter le domaine & services applicatifs**
   - Livrables : entités/valeur/services, validation de règles métier, gestion d'erreurs.
   - CA : UC Must couverts au niveau domaine (tests unitaires verts).
2. **Adapter l'interface (ports/adapters)**
   - Livrables : adapters web (ex. controller), adapters persistance, mapping DTO ↔ domain.
   - CA : séparation nette domain ↔ infra; pas de logique métier dans les controllers.
3. **Exposer des points d'entrée internes (pré-API publique)**
   - Livrables : endpoints internes ou CLI pour enchaîner les UC Must (prototypage rapide).
   - CA : scénario bout-en-bout démontrable dans la VM.

5) Qualité, tests & sécurité de base

1. **Mettre en place la stratégie de tests**
   - Livrables : pyramide de tests (unit → intégration → E2E), coverage minimal ciblé.
   - CA : ≥ 80% domaines critiques; E2E qui orchestre 1 scénario clé.
2. **Sécurité applicative minimale**
   - Livrables : gestion erreurs uniformisée (codes, messages), validation/assainissement des entrées, logs d'accès.
   - CA : pas de secrets en clair; secrets gérés via variables; journalisation des erreurs.

6) CI/CD & conteneurisation (J10–J22)

1. **Conteneuriser l'application & la base**
   - Livrables : `Dockerfile` prod-like (multi-stage), `docker-compose.yml` local/CI, santé (`/health`).
   - CA : `docker compose up` lance app + DB + seed; healthcheck OK.
2. **Mettre en place la CI**
   - Livrables : pipeline (lint → build → tests unit/int/E2E → artefacts), badge CI.

?

- CA : pipeline déterministe < 10 min; fail rapide sur tests KO.

3. **Préparer un job CD (en VM)**
   - Livrables : script de déploiement (compose/swarm/systemd), doc d'exploitation.
   - CA : déploiement en un clic/commande; rollback simple.

7) Documentation finale & démonstration

1. **Finaliser la documentation**
   - Livrables : Documentation en **PDF** avec template Arc42, 4+1 mis à jour, ADRs consolidés, guide d'exploitation (runbook).
   - CA : tout nouveau·elle arrivant·e peut cloner, lancer et tester en < 30 min

Définition de Fini (DoD) — Phase 1

- UC Must implémentés de bout en bout (domaine + persistance + entrée).
- Tests automatisés opérationnelles en CI (unit, intégration, E2E).
- App & DB conteneurisées, déployables sur la VM via un script unique.
- 4+1 **cohérent**, Arc42 (sections cœur), **≥3 ADR** approuvés.
- Logs structurés; guide d'exploitation & de démo à jour.
- Rapport en PDF et projet complet (zip) déposé sur Moodle.

Grille d'évaluation – Projet Phase 1 (BrokerX+)

Critères	Pondération	Excellent (A)	Satisfaisant (B–C)	Insuffisant (D)	Non réalisé (F)
1. Analyse métier & DDD	**15 %**	≥3 cas d'utilisation Must bien décrits (scénarios complets, flux alternatifs), diagrammes de contexte clairs, bounded contexts et glossaire validés	UC partiels (scénarios incomplets), modèle de domaine esquissé mais lacunes de cohérence	UC peu détaillés, modèle de domaine superficiel ou incohérent	Aucun UC documenté, pas de modèle de domaine
2. Architecture & Décisions	**20 %**	Architecture monolithique claire et évolutive, style justifié (Hexagonal/MVC), vues 4+1 complètes, Arc42 sections 1–8 documentées, ≥3 ADR structurés et pertinents	Architecture définie mais justification limitée, vues 4+1 ou Arc42 partielles, 2 ADR seulement ou peu argumentés	Architecture floue, vues manquantes ou incohérentes, décisions non justifiées	Pas d'architecture documentée
3. Persistance & intégrité	**15 %**	Schéma de persistance robuste (UML/ER), migrations reproductibles, DAOs/ORM implémentés et testés, intégrité et rollback validés	Persistance fonctionnelle mais design incomplet (contraintes manquantes, seeds partiels), tests d'intégration minimaux	Persistance fragile, peu fiable, CRUD incomplet	Pas de persistance implémentée
4. Implémentation du prototype	**15 %**	Domaine et services applicatifs bien séparés, règles métier validées, interfaces (adapters) claires, UC Must exécutables bout-en-bout sur VM	Domaine et services fonctionnels mais couplage fort, certains UC Must manquants, séparation domain/infra incomplète	Implémentation partielle ou bugguée, peu de respect DDD/patrons	Aucun prototype fonctionnel
5. Qualité, tests & sécurité	**15 %**	Suite de tests complète (unit, intégration, E2E), couverture >80 % sur domaine critique, scénario E2E automatisé, sécurité minimale (validation entrées, gestion d'erreurs, secrets protégés)	Tests unitaires et intégration présents mais E2E incomplet, couverture <80 %, sécurité partielle	Tests superficiels, absence de stratégie de validation/erreurs	Pas de tests automatisés
6. CI/CD & Conteneurisation	**10 %**	Dockerfile multi-stage, docker-compose complet (app + DB + seed), healthcheck OK, pipeline CI stable <10 min (lint, build, tests), script de déploiement en VM opérationnel avec rollback	Conteneurisation fonctionnelle mais incomplète (pas de healthcheck, doc partielle), pipeline CI fonctionnel mais fragile	Conteneurisation minimale (un seul conteneur), CI/CD partielle ou non reproductible	Pas de conteneurisation ni CI/CD
7. Documentation & démonstration	**10 %**	Rapport final PDF clair et complet (Arc42, 4+1, ADR consolidés, runbook), guide de démo cohérent, reproductibilité <30 min sur VM	Rapport présent mais sections incomplètes, guide de démo sommaire, reproductibilité partielle	Rapport superficiel, peu exploitable, démo difficile	Aucun rapport/ démo fournie

?

Barème indicatif

- **A (85–100 %)** : Projet complet, robuste et bien documenté. Toutes les exigences critiques respectées.
- **B (70–84 %)** : Projet solide mais avec quelques lacunes dans la documentation, les tests ou la persistance.
- **C (60–69 %)** : Projet fonctionnel mais partiel, couverture faible et justification limitée.
- **D (50–59 %)** : Projet incomplet, architecture fragile ou mal documentée.
- **F (<50 %)** : Projet non fonctionnel, livrables absents ou incohérents.

Bon travail!

Ajouter un travail

Statut de remise

Statut des travaux remis	Aucun devoir n'a encore été remis
Statut de l'évaluation	Non évalué
Temps restant	26 jours 12 heures restants

?