

DOCUMENT TECHNIQUE

Refonte de l'Application Legacy

**Date :**

**26/10/2025**

**Préparé par :**

**Bilel Jandoubi & Mounir Oueslati**

**Projet :**

**Migration JSF/JSP  Angular 19 & Spring Boot**

**1. CONTEXTE ET OBJECTIFS**

Ce document présente une analyse détaillée des problèmes identifiés dans l'application actuelle développée en JSF et JSP, ainsi que les solutions proposées dans le cadre de la refonte technologique vers Angular 19 et Spring Boot. L'objectif est de fournir un référentiel technique pour l'évaluation et la sélection d'un consultant externe compétent.

**2. ARCHITECTURE SYSTÈME**

# Problème : Architecture Monolithique

* + - Constat : L'application JSF/JSP actuelle présente une architecture monolithique avec un couplage fort entre les couches présentation, métier et données. Cela limite la scalabilité et complique la maintenance.
* Solution Proposée : Adopter une architecture modulaire avec Spring Boot permettant une séparation claire des préoccupations (Separation of Concerns). Mise en place d'une architecture en couches (Controller, Service, Repository) avec des interfaces bien définies. Possibilité d'évolution vers une architecture microservices si nécessaire.

# Problème : Gestion des Sessions Serveur

* + - Constat : JSF utilise une gestion de session côté serveur lourde (Stateful), consommant beaucoup de ressources mémoire et limitant la scalabilité horizontale.
* Solution Proposée : Application REST stateless avec Angular 19. Authentification par JWT (JSON Web Tokens) permettant une gestion de session côté client. Cela facilite le load balancing et améliore les performances globales du système.

# Problème : Dépendances Obsolètes

* + - Constat : Les technologies JSF et JSP sont en déclin avec un support communautaire limité et des vulnérabilités de sécurité non corrigées. Difficulté à recruter des développeurs maîtrisant ces technologies legacy.
* Solution Proposée : Migration vers Angular 19 (framework moderne avec support LTS) et Spring Boot (écosystème actif et mature). Ces technologies bénéficient d'une large communauté, de mises à jour régulières et d'un vivier de talents disponible.

**3. PROBLÈMES DE SYSTEM DESIGN**

# Problème : Couplage Fort entre Composants

* + - Constat : Dépendances directes entre modules créant un effet domino lors des modifications. Impossible de modifier ou remplacer un composant sans impacter l'ensemble du système. Difficulté à réutiliser le code dans d'autres contextes.
* Solution Proposée : Implémentation des principes SOLID avec injection de dépendances Spring. Utilisation d'interfaces pour définir les contrats entre composants. Pattern Repository pour l'accès aux données. DTOs (Data Transfer Objects) pour découpler les entités métier des API. Événements Spring pour la communication asynchrone entre modules.

# Problème : Absence de Séparation Frontend/Backend

* + - Constat : Logique métier mélangée dans les pages JSP. Impossible de développer frontend et backend indépendamment. Difficulté pour créer des applications mobiles ou des intégrations tierces.
* Solution Proposée : Architecture découplée avec API REST. Backend Spring Boot exposant des endpoints RESTful bien documentés (OpenAPI/Swagger). Frontend Angular 19 consommant les APIs. Contrat d'interface clair permettant des développements parallèles. Possibilité d'avoir plusieurs clients (web, mobile, partenaires) utilisant la même API.

# Problème : Gestion d'État Complexe

* + - Constat : État de l'application dispersé entre sessions serveur, cookies et variables JavaScript. Difficultés de debugging et incohérences d'état. Perte de données lors de rafraîchissements.
* Solution Proposée : Gestion d'état centralisée côté client avec NgRx ou Akita pour Angular. Store Redux pattern pour un état prévisible et traçable. Persistance sélective dans LocalStorage. Synchronisation claire avec le backend via APIs. DevTools pour débugger l'état de l'application.

# Problème : Absence de Stratégie de Cache

* + - Constat : Requêtes répétitives vers la base de données pour les mêmes données. Charge inutile sur le serveur et temps de réponse dégradés. Pas de cache distribué pour la scalabilité horizontale.
* Solution Proposée : Mise en place de stratégies de cache multi-niveaux. Cache L1 avec @Cacheable Spring. Cache L2 distribué avec Redis pour les sessions et données partagées. Cache HTTP côté client avec headers appropriés (ETag, Cache-Control). Cache applicatif Angular avec intercepteurs HTTP. Invalidation intelligente du cache basée sur les événements métier.

# Problème : Gestion des Erreurs Inadaptée

* + - Constat : Messages d'erreur techniques exposés aux utilisateurs. Pas de distinction entre erreurs techniques et erreurs métier. Absence de mécanisme de retry et de fallback. Logging insuffisant pour le debugging.
* Solution Proposée : Gestion centralisée des exceptions avec @ControllerAdvice Spring. Codes d'erreur standardisés et messages utilisateur clairs. Pattern Circuit Breaker avec Resilience4j pour les services externes. Retry automatique avec backoff exponentiel. Logging structuré avec SLF4J/Logback incluant trace ID pour corréler les logs.

# Problème : Monitoring et Observabilité Limités

* + - Constat : Absence de métriques applicatives en temps réel. Difficultés à identifier les goulots d'étranglement. Pas de traçabilité des requêtes à travers le système. Détection tardive des problèmes en production.
* Solution Proposée : Intégration Spring Boot Actuator pour health checks et métriques. Prometheus pour la collecte de métriques (temps de réponse, taux d'erreur, throughput). Grafana pour la visualisation en temps réel. Distributed tracing avec Zipkin ou Jaeger. Logs centralisés avec ELK Stack (Elasticsearch, Logstash, Kibana). Alerting proactif basé sur les SLAs.

# Problème : Scalabilité Limitée (Verticale uniquement)

* + - Constat : Architecture ne permettant que la scalabilité verticale (augmentation des ressources). Sessions serveur empêchant la distribution de charge efficace. Coût élevé et limites physiques des serveurs. Pas de mécanisme d'auto-scaling.
* Solution Proposée : Architecture stateless permettant la scalabilité horizontale. Load balancing avec NGINX ou HAProxy. Déploiement containerisé avec Docker. Orchestration Kubernetes pour auto-scaling basé sur la charge (CPU, mémoire, requêtes/sec). Séparation des composants stateful (base de données) et stateless (application). Configuration pour Cloud deployment (AWS, Azure, GCP).

# Problème : Déploiement Monolithique Complexe

* + - Constat : Déploiement all-or-nothing nécessitant l'arrêt complet de l'application. Temps de déploiement long et fenêtres de maintenance étendues. Rollback complexe en cas de problème. Impossible de déployer des corrections rapidement.
* Solution Proposée : Pipeline CI/CD automatisé avec GitLab CI ou Jenkins. Déploiement Blue-Green ou Canary pour zero-downtime. Tests automatisés (unitaires, intégration, e2e) avant déploiement. Rollback automatique en cas d'échec des health checks. Containerisation permettant des déploiements cohérents entre environnements. Feature flags pour activation progressive des fonctionnalités.

# Problème : Gestion des Configurations

* + - Constat : Fichiers de configuration embarqués dans l'application. Nécessité de recompiler pour changer la configuration. Configurations différentes difficiles à gérer entre environnements. Secrets (mots de passe, clés API) stockés en clair dans le code.
* Solution Proposée : Externalisation des configurations avec Spring Cloud Config. Profils Spring pour les différents environnements (dev, test, prod). Variables d'environnement pour les paramètres sensibles. Secrets management avec HashiCorp Vault ou AWS Secrets Manager. Configuration rechargeable à chaud via Spring Boot Actuator. Versioning des configurations dans Git.

# Problème : Design d'API Incohérent

* + - Constat : Endpoints non standardisés avec conventions de nommage variables. Absence de versioning d'API. Pagination, filtrage et tri non uniformes. Absence de documentation à jour des APIs.
* Solution Proposée : Respect des conventions REST (HTTP verbs, status codes, resource naming). Versioning d'API via URL (/api/v1/...) ou headers. Standards de pagination avec Spring Data (page, size, sort). Documentation automatique avec Swagger/OpenAPI 3.0. Validation des inputs avec Bean Validation (@Valid, @Validated). HATEOAS pour APIs auto-descriptives si nécessaire. Rate limiting pour protéger les APIs.

# Problème : Traitement Asynchrone Absent

* + - Constat : Tous les traitements sont synchrones, bloquant l'utilisateur. Timeouts fréquents pour les opérations longues. Pas de système de file d'attente. Impossible de traiter les pics de charge.
* Solution Proposée : Implémentation de traitements asynchrones avec @Async Spring. File de messages avec RabbitMQ ou Apache Kafka pour les traitements lourds. Jobs batch avec Spring Batch pour les traitements planifiés. WebSockets ou Server-Sent Events pour les notifications en temps réel. Pattern Saga pour les transactions distribuées si microservices.

# Problème : Absence de Stratégie de Backup et Recovery

* + - Constat : Backup de base de données manuel et irrégulier. Pas de plan de disaster recovery. Recovery Time Objective (RTO) et Recovery Point Objective (RPO) non définis. Pas de tests de restauration.
* Solution Proposée : Backup automatisé quotidien de la base Oracle avec RMAN. Snapshots des volumes de données. Réplication de base de données pour haute disponibilité. Documentation du plan de disaster recovery avec procédures testées. Définition claire des RTO/RPO selon les besoins métier. Tests réguliers de restauration sur environnement de staging.

**4. BEST PRACTICES POUR APPLICATIONS BANCAIRES**

Les applications bancaires nécessitent des standards de qualité, sécurité et fiabilité particulièrement élevés. Cette section présente les meilleures pratiques reconnues dans l'industrie bancaire qui doivent être implémentées dans la refonte de l'application.

# Sécurité et Conformité Réglementaire

PCI-DSS (Payment Card Industry Data Security Standard) : Toute application bancaire manipulant des données de cartes bancaires doit respecter les 12 exigences PCI-DSS. Chiffrement end-to-end des données sensibles, segmentation réseau, tests de pénétration réguliers, et journalisation détaillée de tous les accès aux données cardholder.

RGPD et Protection des Données : Mise en œuvre des principes Privacy by Design. Chiffrement AES-256 au repos et TLS 1.3 en transit. Pseudonymisation et anonymisation des données personnelles. Droit à l'oubli et portabilité des données. Registre des traitements et analyse d'impact DPIA pour les traitements à risque.

Authentification Forte (PSD2) : Authentification multi-facteurs obligatoire conforme à la directive PSD2. Combinaison de facteurs : connaissance (mot de passe), possession (token, SMS OTP), et inhérence (biométrie). Dynamic linking pour les paiements. Exemptions SCA selon le niveau de risque transactionnel.

# Architecture Haute Disponibilité

SLA de 99.99% (Four Nines) : Architecture redondante avec élimination des single points of failure. Load balancing actif-actif sur plusieurs zones de disponibilité. Basculement automatique (failover) en moins de 30 secondes. Downtime maximum acceptable : 52 minutes par an.

Base de Données Multi-Maître : Réplication Oracle RAC (Real Application Clusters) ou Data Guard pour la haute disponibilité. Synchronisation synchrone pour les transactions critiques. Réplication asynchrone vers sites de secours distants. Sharding horizontal pour distribuer la charge sur plusieurs nœuds.

Design for Failure : Assomption que tout composant peut tomber en panne. Circuit breaker pattern pour isoler les défaillances. Timeouts agressifs et retry avec backoff exponentiel. Graceful degradation permettant le fonctionnement en mode dégradé plutôt qu'une panne complète.

# Audit et Traçabilité Complète

Audit Trail Immutable : Journalisation de toutes les opérations bancaires dans une base d'audit séparée et immutable. Enregistrement de qui, quoi, quand, où, et pourquoi. Horodatage signé cryptographiquement. Conservation pendant minimum 10 ans pour conformité réglementaire. Possibilité de reconstituer l'état du système à tout moment (event sourcing).

Traçabilité des Transactions : Identifiant unique (UUID) pour chaque transaction propagé à travers tous les systèmes. Correlation ID permettant de suivre une opération de bout en bout. Logs structurés JSON avec contexte complet (utilisateur, IP, device fingerprint). Capacité de replay des transactions pour analyse forensique.

Monitoring des Accès Privilégiés : Surveillance en temps réel des comptes administrateurs. Principe du moindre privilège strictement appliqué. Accès just-in-time avec élévation temporaire. Enregistrement vidéo des sessions administrateur. Alerting immédiat sur comportements suspects.

# Architecture Transactionnelle Robuste

Propriétés ACID Strictes : Atomicité garantie pour toutes les transactions financières. Isolation au niveau Serializable pour éviter les anomalies. Durabilité avec écriture sur disque avant commit. Utilisation de 2PC (Two-Phase Commit) pour les transactions distribuées. Gestion des deadlocks avec retry intelligent.

Idempotence des Opérations : Toutes les API bancaires doivent être idempotentes. Déduplication basée sur idempotency key unique. Protection contre les doubles soumissions. Cache des résultats pour garantir la cohérence. Crucial pour la fiabilité des paiements et virements.

Compensation Transactions (Saga Pattern) : Pour les processus multi-étapes, implémentation de transactions compensatoires. En cas d'échec, rollback automatique via opérations inverses. État transactionnel persisté pour reprise après panne. Gestion de la cohérence éventuelle pour systèmes distribués.

# Séparation des Environnements et Données

Environnements Isolés : Séparation stricte Dev / QA / Staging / Production avec réseaux isolés. Production jamais accessible directement depuis internet. Bastion hosts pour accès administratif sécurisé. Aucune donnée de production en environnements inférieurs.

Data Masking et Tokenization : Anonymisation automatique des données sensibles pour les environnements non-production. Tokenization des numéros de carte et comptes bancaires. Format Preserving Encryption (FPE) pour conserver la structure des données. Vault centralisé pour gestion des secrets et clés de chiffrement.

Zero Trust Architecture : Principe du 'Never Trust, Always Verify'. Micro-segmentation réseau avec contrôles d'accès entre services. mTLS (mutual TLS) pour authentification service-to-service. Vérification continue de l'identité et du contexte. Principe du moindre privilège appliqué à tous les niveaux.

# Gestion des Performances et Capacité

SLO et SLI Définis : Service Level Objectives basés sur des métriques métier. Temps de réponse < 200ms pour 95% des requêtes critiques. Throughput minimum garanti (ex: 1000 transactions/seconde). Error budget pour équilibrer innovation et stabilité. Dégradation gracieuse si SLO non respectés.

Capacity Planning Proactif : Modélisation de la croissance basée sur historique. Auto-scaling horizontal basé sur CPU, mémoire et métriques custom. Load testing régulier simulant pics de charge (fin de mois, périodes promotionnelles). Réserve de capacité 30% pour absorber les pics imprévus.

Rate Limiting et Throttling : Protection contre abus et attaques DDoS. Limites par utilisateur, par IP, et par API endpoint. Token bucket ou leaky bucket algorithms. Réponses HTTP 429 avec Retry-After header. Whitelist pour services internes et partenaires.

# Disaster Recovery et Business Continuity

RPO et RTO Bancaires : Recovery Point Objective < 5 minutes pour données critiques. Recovery Time Objective < 1 heure pour services essentiels. Backup géographiquement distribué sur minimum 3 sites distants de 100+ km. Réplication synchrone vers site primaire de secours, asynchrone vers sites secondaires.

Plan de Continuité Testé : Exercices de disaster recovery trimestriels documentés. Runbooks détaillés pour chaque scénario de panne. Équipes on-call 24/7 avec escalade définie. Tests de basculement sans impact sur production. War rooms virtuels prêts à l'emploi.

Stratégie de Backup 3-2-1 : 3 copies des données (production + 2 backups). 2 types de média différents (disque, bande, cloud). 1 copie off-site géographiquement distant. Chiffrement AES-256 de tous les backups. Tests de restauration mensuels sur environnement isolé. Immutabilité des backups (WORM - Write Once Read Many).

# Détection et Prévention de la Fraude

Analyse Comportementale en Temps Réel : Machine Learning pour détecter patterns anormaux. Scoring de risque pour chaque transaction. Blocage automatique ou demande de confirmation pour opérations suspectes. Analyse de velocity (nombre d'opérations/temps). Géolocalisation incohérente détectée (impossible travel).

Device Fingerprinting : Identification unique de chaque appareil. Détection de changement de device sans notification. Liste blanche des devices de confiance. Détection d'émulateurs et environnements rootés/jailbreakés. Challenge supplémentaire pour nouveaux devices.

Listes Noires et Sanctions : Vérification automatique contre listes OFAC, ONU, UE. Screening des noms et pays en temps réel. Blocage des transactions vers juridictions à haut risque. Alertes pour tentatives d'accès depuis IP malveillantes connues. Conformité AML (Anti-Money Laundering) et KYC (Know Your Customer).

# Architecture par Domaines Métier

Domain-Driven Design (DDD) : Organisation du code par domaines bancaires (Comptes, Virements, Prêts, Cartes, etc.). Bounded contexts clairement définis avec leur propre modèle de données. Ubiquitous language partagé entre développeurs et métier. Agrégats garantissant la cohérence transactionnelle.

Event-Driven Architecture : Communication asynchrone via événements métier. Event store comme source de vérité (Event Sourcing). Reconstruction de l'état via replay d'événements. CQRS (Command Query Responsibility Segregation) pour optimiser lectures et écritures. Kafka ou RabbitMQ pour messagerie fiable.

API Gateway Pattern : Point d'entrée unique pour tous les clients. Authentification, autorisation, et rate limiting centralisés. Agrégation de requêtes et transformation de données. Versioning et routage intelligent. Cache et compression. Analytics et monitoring du trafic API.

# Conformité et Gouvernance

Standards ISO Bancaires : Conformité ISO 27001 (sécurité de l'information). ISO 22301 (business continuity). ISO 20022 pour messages financiers standardisés. Audits externes réguliers et certifications. Documentation des contrôles COBIT et ITIL.

Ségrégation des Tâches (SoD) : Séparation stricte des rôles critiques. Un développeur ne peut pas déployer son propre code. Approbations multiples pour changements en production. Principe des quatre yeux pour opérations sensibles. Matrices RACI documentées et auditables.

Change Management Rigoureux : Tous changements passent par CAB (Change Advisory Board). Documentation obligatoire avec risk assessment. Fenêtres de maintenance planifiées. Rollback plan validé avant chaque déploiement. Post-implementation review systématique. Approbations traçables dans système de ticketing.

# Optimisation des Coûts Cloud

FinOps et Cost Management : Tagging systématique des ressources par projet/département. Monitoring temps réel des coûts avec alerting sur seuils. Réservations d'instances pour charges prévisibles (-50% coûts). Auto-scaling down en heures creuses. Archivage automatique des données froides vers tiers économiques.

Architecture Multi-Cloud Selective : Services critiques en multi-cloud pour éviter vendor lock-in. Best-of-breed : choisir le meilleur service par provider. Cloud bursting : débordement vers cloud en cas de pic. Abstraction via Kubernetes pour portabilité. Stratégie de sortie documentée pour chaque service cloud.

# Tableau Récapitulatif des Best Practices Prioritaires

|  |  |  |
| --- | --- | --- |
| **Catégorie** | **Best Practice** | **Priorité** |
| Sécurité | Authentification multi-facteurs (PSD2) | **CRITIQUE** |
| Sécurité | Chiffrement end-to-end AES-256/TLS 1.3 | **CRITIQUE** |
| Disponibilité | Architecture redondante multi-zones (99.99%) | **CRITIQUE** |
| Audit | Audit trail immutable toutes transactions | **CRITIQUE** |
| Transactions | Idempotence + propriétés ACID strictes | **CRITIQUE** |
| Data | Backup 3-2-1 avec RPO < 5min | **CRITIQUE** |
| Conformité | PCI-DSS + RGPD + ISO 27001 | **ÉLEVÉE** |
| Fraude | ML pour détection temps réel | **ÉLEVÉE** |
| Performance | SLO définis (95% < 200ms) | **ÉLEVÉE** |
| Architecture | DDD + Event-Driven + API Gateway | MOYENNE |

**5. DESIGN ET EXPÉRIENCE UTILISATEUR**

* 1. **Problème : Interface Non Responsive**
     + Constat : L'interface JSF n'est pas adaptée aux appareils mobiles et tablettes. Expérience utilisateur dégradée sur différentes tailles d'écran.
* Solution : Application Angular 19 responsive avec Angular Material. Design mobile-first.

# Problème : Performance Frontend

* + - Constat : Rechargements complets de page. Temps de réponse lents.
* Solution : SPA avec lazy loading, cache intelligent, navigation instantanée.

# Problème : Composants UI Limités

* + - Constat : Composants JSF peu personnalisables.
* Solution : Angular Material, PrimeNG avec composants modernes et accessibles.

**6. BASE DE DONNÉES ORACLE**

# Problème : Requêtes SQL Embarquées

* + - Constat : SQL natif mélangé au code. Couplage fort. Risques SQL Injection.
* Solution : Spring Data JPA avec Hibernate. Repository pattern. Requêtes typées.

# Problème : Gestion Manuelle des Transactions

* + - Constat : Gestion programmatique dispersée. Risques d'incohérence.
* Solution : @Transactional Spring. Gestion déclarative. HikariCP.

# Problème : Performance des Requêtes

* + - Constat : Problèmes N+1, absence de pagination.
* Solution : Fetch optimisé, pagination native, projections DTO, cache Redis.

# Problème : Migration du Schéma

* + - Constat : Pas de versioning du schéma.
* Solution : Flyway/Liquibase pour migrations versionnées.

**7. SÉCURITÉ**

# Problème : Authentification Legacy

* + - Constat : Authentification sessions serveur. Pas de 2FA.
* Solution : Spring Security + JWT. OAuth2/OpenID Connect. 2FA.

# Problème : Gestion des Autorisations

* + - Constat : Contrôles d'accès dispersés.
* Solution : @PreAuthorize, @Secured. RBAC/ABAC centralisé.

**8. MAINTENANCE ET TESTABILITÉ**

# Problème : Tests Limités

* + - Constat : Code difficile à tester. Pas de tests automatisés.
* Solution : JUnit 5, Mockito, @SpringBootTest. Tests Angular (Jasmine/Karma). CI/CD.

# Problème : Documentation Technique

* + - Constat : Documentation inexistante ou obsolète.
* Solution : Swagger/OpenAPI. Compodoc pour Angular. Code auto-documenté.

**9. RECOMMANDATIONS POUR LE CONSULTANT**

Le consultant sélectionné devra démontrer une expertise solide dans les domaines suivants :

**Expertise Technique** • Maîtrise d'Angular 19 (version LTS)

* + Expertise Spring Boot 3.x et écosystème Spring
  + Connaissance approfondie d'Oracle Database
  + Expérience avec JPA/Hibernate

**Architecture** • Conception d'architectures REST/RESTful

* + Patterns de conception (Repository, Service, etc.)
  + Microservices
  + Sécurité applicative (Spring Security, JWT)

**Migration** • Expérience migrations JSF/JSP vers moderne

* + Stratégies de migration progressive
  + Gestion des données et schéma
  + Formation des équipes

**Qualité & DevOps** • Tests automatisés (unitaires, intégration, e2e)

* + CI/CD (GitLab, Jenkins)
  + Revue de code et bonnes pratiques
  + Documentation technique

**Bancaire** • Expérience applications bancaires

* + Connaissance PCI-DSS, RGPD, PSD2
  + Architecture haute disponibilité
  + Sécurité et audit trail

**10. CONCLUSION**

La migration de l'application legacy JSF/JSP vers une architecture moderne Angular 19 / Spring Boot représente une opportunité stratégique pour la BNA. Cette refonte permettra de résoudre les problèmes actuels d'architecture, de performance, de maintenabilité et de sécurité, tout en préparant le système pour les évolutions futures.

Le succès de ce projet repose sur la sélection d'un consultant expérimenté capable non seulement de maîtriser les technologies cibles, mais aussi de comprendre les enjeux métier de la BNA et d'accompagner les équipes dans cette transformation digitale.

Document Confidentiel - Banque Nationale Agricole (BNA) - 27/10/2025