



UNIVERSITÉ DE VERSAILLES  
SAINT-QUENTIN-EN-YVELINES

UNIVERSITÉ PARIS SACLAY

MASTER 2 DATA SCALE

---

# Service Web Composite : Évaluation de Demande de Prêt Immobilier

---

**Étudiants :**

Sarah HARROUCHE

Boualem MOKEDDEM

Souhil OUCHENE

Année Universitaire 2025-2026

# Table des matières

<b>1</b>	<b>Introduction</b>	<b>3</b>
<b>2</b>	<b>Séparation CRUD vs Logique Métier</b>	<b>4</b>
2.1	Justification Architecturale . . . . .	4
2.1.1	Réutilisabilité . . . . .	4
2.1.2	Maintenabilité . . . . .	4
2.1.3	Testabilité . . . . .	4
2.2	Implémentation . . . . .	4
<b>3</b>	<b>Orchestration (Processus BPMN)</b>	<b>6</b>
3.1	Chaînage des Services . . . . .	6
3.2	Diagramme BPMN . . . . .	6
<b>4</b>	<b>Choix de Conception</b>	<b>7</b>
4.1	Binding SOAP : Document/Literal Wrapped . . . . .	7
4.1.1	Justification . . . . .	7
4.1.2	Impact technique . . . . .	7
4.2	Protocole SOAP 1.1 . . . . .	8
4.3	Architecture Synchrone vs Asynchrone . . . . .	8
<b>5</b>	<b>Gestion des Fautes et Versioning</b>	<b>9</b>
5.1	Stratégie de Mappage SOAP Fault → HTTP . . . . .	9
5.1.1	Exemple : Client Non Trouvé . . . . .	9
5.2	Versioning : Namespace-Based . . . . .	9
<b>6</b>	<b>Tests et Validation</b>	<b>11</b>
6.1	Détails Techniques des Tests Unitaires . . . . .	11
6.1.1	Framework et Configuration . . . . .	11
6.1.2	Cas de Test Critiques . . . . .	11
6.2	Détails Techniques des Tests d'Intégration . . . . .	12
6.2.1	Framework et Setup . . . . .	12
6.2.2	Cas End-to-End . . . . .	12
<b>7</b>	<b>Mesures SLA/QoS</b>	<b>13</b>

7.1 Métriques Réelles (Docker Windows) . . . . . 13

**8 Conclusion** . . . . . **14**

# Chapitre 1

## Introduction

Ce projet implémente un service web composite basé sur une architecture SOA pour évaluer la solvabilité d'un demandeur de crédit immobilier. L'architecture intègre huit micro-services SOAP orchestrés de manière synchrone, respectant les standards d'interopérabilité et de maintenabilité définis pour un système d'entreprise.

# Chapitre 2

## Séparation CRUD vs Logique Métier

### 2.1 Justification Architecturale

L'architecture applique le pattern **Separation of Concerns** en distinguant explicitement les services de **lecture de données (CRUD)** des services de **logique métier**. Cette séparation répond à trois objectifs fondamentaux :

#### 2.1.1 Réutilisabilité

Les services CRUD (`ClientDirectoryService`, `FinancialDataService`, `CreditBureauService`) ne contiennent aucune logique métier. Ils exposent uniquement des opérations de lecture typées : `GetClientIdentity()`, `GetClientFinancials()`, `GetClientCreditHistory()`. Cette abstraction permet à d'autres domaines (ex. assurance, scoring d'autres produits) de réutiliser ces mêmes services sans dépendre de la logique de crédit immobilier.

#### 2.1.2 Maintenabilité

Un bug ou une modification dans la base de données clients n'impacte que le service CRUD correspondant, sans affecter les services métier. Inversement, un changement de règle métier (ex. nouveau seuil de solvabilité) modifie uniquement `SolvencyDecisionService`, sans toucher aux accès données. Cette isolation réduit les risques de régression.

#### 2.1.3 Testabilité

Les services métier peuvent être testés unitairement en mockant simplement les réponses CRUD. Par exemple, tester le calcul de score ne nécessite pas une base de données complète : il suffit de passer des paramètres (dette, retards, faillite) directement au service. Les tests d'intégration valident ensuite le chaînage complet.

### 2.2 Implémentation

**Services CRUD** (Port 5002, namespace : `crud:v1`) :

- Dépendance : aucune
- État : stateless (lecture seule)
- Faults : `Client.NotFound`, `Client.ValidationError`

**Services Métier** (Ports 5003, 5005, 5007, namespaces `:business:v1`, `:appraisal:v1`, `:approval:v1`) :

- Dépendance : services CRUD + autres métier
- État : calculs déterministes (pas de side-effects)
- Faults : `Business.ScoringError`, `Approval.DecisionError`

Cette séparation a permis de développer les deux catégories de services en parallèle, puis de les intégrer progressivement via l'orchestrateur.

# Chapitre 3

## Orchestration (Processus BPMN)

### 3.1 Chaînage des Services

L'orchestrateur (Port 5004) coordonne l'exécution synchrone de sept étapes en séquence :

1. **Extraction** (IE :5006) : parse le texte de demande, extrait champs structurés
2. **Accès données** (CRUD :5002) : récupère profil, financières, historique crédit
3. **Scoring** (Business :5003) : applique formule de score
4. **Solvabilité** (Business :5003) : vérifie seuil score  $\geq 700$  ET revenus  $>$  dépenses
5. **Évaluation propriété** (Appraisal :5005) : estime valeur via comparables de marché
6. **Approbation** (Approval :5007) : décide prêt selon LTV, DTI, risque
7. **Notification** (Notification :5008) : envoie email client (best-effort async)

### 3.2 Diagramme BPMN

*[À insérer : Diagramme BPMN montrant les 8 étapes d'orchestration avec Gateway de décision apres solvabilité]*

**Points clés du processus :**

- Arrêt immédiat si Client.NotFound : retour erreur 404
- Gateway logique : si NOT solvent  $\rightarrow$  approbation directe rejetée
- Enchaînement synchrone garantit traçabilité via correlation ID
- Notification en mode best-effort (ne bloque pas décision)

# Chapitre 4

## Choix de Conception

### 4.1 Binding SOAP : Document/Literal Wrapped

#### 4.1.1 Justification

Trois styles SOAP existent : RPC/encoded, RPC/literal, document/literal. Le choix de **document/literal wrapped** répond à un critère d'interopérabilité :

- **RPC/encoded** : considéré obsolète par le W3C depuis 2005, problèmes interop multi-langages
- **RPC/literal** : interop moyen, moins de flexibilité pour versioning
- **document/literal wrapped** : conforme WS-I Basic Profile, statut de facto pour SOAP d'entreprise

Le binding document/literal wrapped signifie :

- Chaque opération génère une classe wrapper contenant les paramètres (ex. `ComputeCreditScoreRequest`),
- Les types sont définis explicitement en XSD, pas dérivés du WSDL,
- Chaque paramètre devient un élément XML nommé, facilitant l'évolution.

#### 4.1.2 Impact technique

Ce choix impose de définir des ComplexTypes XSD plutôt que des éléments simples. Par exemple, pour retourner un score + grade, nous créons :

```
<complexType name="CreditScore">
  <sequence>
    <element name="score" type="integer"/>
    <element name="grade" type="string"/>
  </sequence>
</complexType>
```

plutôt que de retourner une string "800,A". Cela garantit la validation côté serveur et client, et rend le contrat stable pour versioning futur.

## 4.2 Protocole SOAP 1.1

SOAP 1.2 n'a pas apporté de bénéfice fonctionnel majeur pour ce cas d'usage, tandis que SOAP 1.1 bénéficie d'un support plus large en outils (SoapUI, Zeep) et en documentation. Le choix SOAP 1.1 simplifie la mise en place sans compromis technique.

## 4.3 Architecture Synchrone vs Asynchrone

Les services sont orchestrés de manière **synchrone** : chaque étape attend la réponse précédente. Cette approche garantit :

- Traçabilité : une erreur à l'étape  $N$  interrompt immédiatement, pas de bruit asynchrone
- Corrélation : un seul correlation ID couvre toute la transaction
- Implémentation simple : pas de message queue, pas de retry complexe

En contrepartie, la latence globale dépend du temps de chaque service (actuellement 1.4s médiane en Docker Windows). Une future optimisation pourrait intégrer du parallélisme partiel (ex. appraisal et approval en parallèle après scoring).

# Chapitre 5

## Gestion des Fautes et Versioning

### 5.1 Stratégie de Mappage SOAP Fault → HTTP

Les services SOAP retournent des `<soap:Fault>` typés avec codes d'erreur métier (ex. `Client.NotFound`). L'adapter REST (Port 5001) traduit ces faults en codes HTTP appropriés :

- **Client.NotFound** → HTTP 404 : Ressource n'existe pas
- **Client.ValidationError** → HTTP 400 : Format invalide, ex. `client-abc`
- **Property.RegionNotFound** → HTTP 202 : Région inconnue, expertise requise
- **Server.OrchestrationError** → HTTP 500 : Erreur interne

**Justification** : Exposer le code métier dans la fault permet au client de discriminer les erreurs (validation client, données incomplètes, indisponibilité réseau) sans parser le message. Le code HTTP classe ensuite l'erreur pour la couche HTTP (4xx = client, 5xx = serveur).

#### 5.1.1 Exemple : Client Non Trouvé

SOAP Fault :

```
<faultcode>Client.NotFound</faultcode>
<faultstring>Client 'client-999' non trouvé</faultstring>
```

Réponse REST Adapter :

```
HTTP 404
{"status": "error", "fault_code": "Client.NotFound",
 "error": "Client 'client-999' non trouvé dans le système"}
```

### 5.2 Versioning : Namespace-Based

Le versioning utilise les namespaces WSDL :

```
v1 : urn:solvency.verifcation.service:v1
v2 : urn:solvency.verifcation.service:v2
```

**Scénario Migration v1→v2** : Ajout d'un champ optionnel `employment_type` aux financières

- v1 clients : reçoivent l'ancienne structure (backward compat)
- v2 clients : peuvent envoyer `employment_type` sans casser v1
- Serveur : expose les deux namespaces simultanément
- Dépréciation : après 6 mois, v1 est retiré

Cette approche évite une migration forcée et permet aux clients de monter en version à leur rythme.

# Chapitre 6

## Tests et Validation

### 6.1 Détails Techniques des Tests Unitaires

#### 6.1.1 Framework et Configuration

Les tests unitaires utilisent **pytest 7.4.0** avec le plugin **pytest-cov** pour la couverture de code. Configuration (pytest.ini) :

```
[pytest]
testpaths = tests/
timeout = 30
addopts = -v --strict-markers --tb=short
```

Couverture cible : 80

#### 6.1.2 Cas de Test Critiques

**Calcul de Score** : Vérification de la formule  $\text{score} = 1000 - 0.1 \times \text{dette} - 50 \times \text{retards} - (\text{faillite?}200 : 0)$

Cas 1 : client-001 (dette=5000, retards=2, bankruptcy=False)  
score = 1000 - 500 - 100 - 0 = 400  
grade = "D"

Cas 2 : client-003 (dette=15000, retards=5, bankruptcy=True)  
score = 1000 - 1500 - 250 - 200 = -950 → clamped to 0  
grade = "D"

**Seuil Solvabilité** : Vérification des conditions ET

Condition : score ≥ 700 ET income > expenses

Cas 1 : score=800, income=5500, expenses=2500 → solvent  
Cas 2 : score=800, income=3000, expenses=3000 → NOT solvent  
Cas 3 : score=600, income=5500, expenses=2500 → NOT solvent

## 6.2 Détails Techniques des Tests d'Intégration

### 6.2.1 Framework et Setup

Les tests d'intégration utilisent **Zeep 4.2.1** comme client SOAP pour invoquer le service orchestrator en réseau. Prérequis : services Docker actifs (healthcheck passant).

```
@pytest.fixture(scope="session")
def soap_client():
    client = SoapClient(wsdl='http://localhost:5004/?wsdl',
                        transport=Transport(timeout=30))
    return client
```

### 6.2.2 Cas End-to-End

**Workflow Complet** : client-002 (Alice) de l'extraction à la notification

Étape 1 : POST /api/loan/apply avec données complètes  
Status: 200

Étape 2 : Vérifier extraction  
extracted\_info.client\_id = "client-002"  
extracted\_info.loan\_amount = 300000

Étape 3 : Vérifier scoring  
credit\_assessment.score = 800  
credit\_assessment.status = "solvent"

Étape 4 : Vérifier approbation  
final\_decision.approved = True  
final\_decision.interest\_rate [2.5, 8.0]

Étape 5 : Vérifier notification  
client\_email = "alice.smith@example.com"  
correlation\_id ""

**Résultat** : 26/26 tests passed, workflow validé du REST adapter au dernier micro-service.

# Chapitre 7

## Mesures SLA/QoS

### 7.1 Métriques Réelles (Docker Windows)

Test de 122 requêtes sur 5 minutes :

Latence :

Min:	1,191 ms	
P50:	1,369 ms	(vs cible <100ms)
P95:	1,954 ms	(vs cible <300ms)
P99:	3,073 ms	(vs cible <500ms)
Max:	3,226 ms	

Disponibilité :

Taux d'erreur :	0.00%	(vs cible <1%)
Disponibilité :	100.00%	(vs cible 99%)

**Facteurs latence** : Base orchestration (8 services) + Docker Windows overhead + SOAP marshalling = 1200ms minimum inévitable. Production Linux estimée 40-50

# Chapitre 8

## Conclusion

L'architecture proposée démontre une implémentation mature de SOA avec séparation des responsabilités, gestion robuste des erreurs, et métriques mesurées. Les choix techniques (document/literal, synchrone, namespace versioning) garantissent l'interopérabilité et la maintenabilité. Les tests valident les formules métier et les workflows critiques. Le système est stable (100