Spécification du Service Web Composite : Évaluation de Demande de Prêt Immobilier

Implementaiton du Service de "vérification de solvabilité (SOAP/WSDL)"

1) Contexte métier

Une banque souhaite **évaluer la solvabilité** d'un client à partir :

- de son **profil** (nom, adresse),
- de ses **données financières** (revenus/dépenses mensuels),
- de son **historique de crédit** (dette courante, retards de paiement, faillite).

Le **service d'orchestration** `SolvencyVerificationService` reçoit un `clientId` et renvoie un **rapport de solvabilité**structuré:

- détails client, données financières, historique de crédit,
- **score de crédit**,
- **statut de solvabilité** (*solvent* / *not solvent*),
- **explications** (pour la décision).

---

2) Architecture SOA attendue

2.1 Services CRUD (lecture données)

- `ClientDirectoryService`
  - `GetClientIdentity(clientId) -> {name, address}`
- `FinancialDataService`
  - `GetClientFinancials(clientId) -> {monthlyIncome, monthlyExpenses}`
- `CreditBureauService`
  - `GetClientCreditHistory(clientId) -> {debt, latePayments, hasBankruptcy}`

Les implémentations peuvent être **in-memory** (dictionnaires) mais l'interface doit être propre et **typée**.

2.2 Services métier

- `CreditScoringService`
  - `ComputeCreditScore(clientId, debt, latePayments, hasBankruptcy) -> {score}`

    - **Formule minimale** imposée :
      `score = 1000 - 0.1*debt - 50*latePayments - (hasBankruptcy ? 200 : 0)`
- `SolvencyDecisionService`
    - `DecideSolvency(monthlyIncome, monthlyExpenses, score) -> {status}`
      Règle minimale : *solvent* si `score ≥ 700` **et** `monthlyIncome > monthlyExpenses`.
- `ExplanationService`
    - `Explain(score, monthlyIncome, monthlyExpenses, debt, latePayments, hasBankruptcy) -> {creditScoreExplanation, incomeVsExpensesExplanation, creditHistoryExplanation}`

2.3 Service d'orchestration

- `SolvencyVerificationService`
    - `VerifySolvency(clientId) -> SolvencyReport`

Important : L'orchestrateur **n'implémente pas la logique métier**, il **compose** les résultats des services CRUD et métier.

---

3) Contrat de service (WSDL + XSD)

3.1 Modèles de données (XSD)

Définir dans un XSD séparé (importé par le WSDL) :

- `ClientId` (pattern : `client-\d{3}`),
- `ClientIdentity` (name, address),
- `Financials` (monthlyIncome: decimal ≥0, monthlyExpenses: decimal ≥0),
- `CreditHistory` (debt: decimal ≥0, latePayments: nonNegativeInteger, hasBankruptcy: boolean),
- `CreditScore` (integer, 0–1000),
- `SolvencyStatus` (enum: `solvent` | `not_solvent`),
- `Explanations` (3 chaînes non vides),
- `SolvencyReport` (agrégation des blocs ci-dessus).

3.2 Messages & portTypes

- CRUD/métier : messages requête/réponse par service.
- Orchestration : `VerifySolvencyRequest {clientId}` → `VerifySolvencyResponse {SolvencyReport}`.

3.3 Binding & endpoint

- Protocole : **SOAP 1.1** (ou 1.2 au choix, justifier).
- Style : **document/literal wrapped**.
- Un **endpoint** par service (ou un seul avec opérations distinctes, mais gardez la séparation logique).

3.4 Versioning

- Namespace avec version : `urn:solvency.verification.service:v1`
- Expliquer comment vous versionneriez une V2 (champs optionnels, extension XSD, etc.).

---

4) Données de test (obligatoires)

Implémenter au minimum ce **jeu de données** :

clientId	name	address	income	expenses	debt	late	bankruptcy
client-001	John Doe	123 Main St	4000	3000	5000	2	false
client-002	Alice Smith	456 Elm St	3000	2500	2000	0	false
client-003	Bob Johnson	789 Oak St	6000	5500	10000	5	true

Vérifications attendues (avec la règle minimale) :

- `client-001` → score 400 → **not_solvent**
- `client-002` → score 800 → **solvent**
- `client-003` → score −450 → **not_solvent**

---

5) Exigences fonctionnelles

1. **Vérifier l'existence** du client (sinon **SOAP Fault** `Client.NotFound` avec message explicite).
2. **Valider** les entrées (`clientId` pattern, revenus/dépenses ≥ 0, etc.).
   En cas d'erreur : **SOAP Fault** `Client.ValidationError`.
3. **Retour** de `VerifySolvency` : objet **structuré** conforme au XSD (pas une string de dict).
4. **Idempotence** des lectures CRUD.
5. **Journalisation** (logs) de la corrélation d'appel (tracer `clientId`, timestamps, latences).

---

6) Exigences non-fonctionnelles (QoS/SLA — mini)

- Exposez dans la doc un **SLA pédagogique** : disponibilité cible (ex. 99% en heures de TD), temps de réponse cible (p95 < 300 ms sur données in-memory).
- **QoS** : ajoutez des métriques simples (compteur d'appels par opération, latence moyenne), exposées dans les logs.
- **Sécurité** (niveau TD) : filtrez les caractères dangereux (<, >, `&`) ou activez la validation XML. Mentionnez des pistes (WS-Security) en discussion.

---

7) Implémentation (guideline)

7.1 Étapes

1. **Concevoir le XSD** puis le **WSDL** (import du XSD).
2. Implémenter les **services CRUD** avec data store in-memory.
3. Implémenter les **services métier** (score, décision, explications).
4. Implémenter le **service d'orchestration**.
5. Activer la **validation** d'entrées/sorties (selon stack choisie).
6. **Gérer les Faults** (HTTP 500 + `<soap:Fault>`).
7. **Exposer la WSDL** (`?wsdl`) et vérifier l'**interopérabilité**.
8. Écrire des **tests** (unitaires + intégration via client SOAP).
9. Préparer un **Dockerfile** et un **docker-compose** minimal si nécessaire.
10. Rédiger un **README** exécutable.

7.2 Contraintes techniques minimales

- Style **document/literal**.
- **Types XSD** bien définis (pas d'`xsd:anyType` pour le rapport).
- **Namespaces** cohérents (WSDL & XSD).
- **Journalisation** côté serveur (au moins INFO + erreurs).

---

8) Tests & validation

8.1 Cas nominaux

- `VerifySolvency(client-002)` → `status=solvent` et score 800.
- `VerifySolvency(client-001)` → `status=not_solvent`.

8.2 Cas d'erreur

- `clientId` inconnu → Fault `Client.NotFound`.
- `clientId` invalide (ex. `abc`) → Fault `Client.ValidationError`.

8.3 Outils de test (au choix)

- **SoapUI** : importer la WSDL, générer requêtes, asserter contenu réponse.
- **Python/Zeep** : mini-script client avec 3 appels + asserts.
- **Postman** (mode SOAP) : requêtes XML + assertions.

8.4 Tests unitaires (exemples)

- `ComputeCreditScore`: jeux (debt, late, bankruptcy) → score attendu.
- `DecideSolvency`: (income, expenses, score) → statut attendu.
- `Explain`: non-vide, cohérent avec score/statut.

---

9) Livrables (rendus)

1. **Code source** (services CRUD, métier, orchestration).
2. **WSDL** + **XSD** (fichiers séparés, versionnés).
3. **Dockerfile** (+ optionnel `docker-compose.yml`).
4. **Jeu de tests** :
   - unitaires (framework du langage),
   - intégration (SoapUI project exporté **ou** script Zeep/JAX-WS client).
5. **README.md** (clair et exécutable) :
   - prérequis, build, run, URL du service, URL de la WSDL,
   - exemples de requêtes/réponses,
   - stratégie de tests,
   - SLA/QoS (cibles),
   - limites, axes d'amélioration (WS-Security, persistance réelle, monitoring).
6. **Courte note d'architecture** (2–3 pages) :
   - séparation CRUD vs métier,
   - orchestration (diagramme simple **BPMN** recommandé),
   - choix de conception (binding, style),
   - gestion des fautes, versioning.

---

10) Barème (sur 20)

- (4 pts) **Contrat** : WSDL/XSD propres, document/literal, namespaces & versioning.
- (5 pts) **Implémentation** : séparation CRUD/métier/orchestration, validation, Faults.
- (3 pts) **Tests** : unitaires + intégration automatisée (SoapUI/Zeep) avec assertions.
- (2 pts) **Qualité** : logs utiles, structure du projet, lisibilité, commentaires.
- (2 pts) **README** : reproductible, exemples, endpoints, WSDL, instructions claires.
- (2 pts) **SLA/QoS** : cibles définies + métriques simples (au moins logs).
- (2 pts) **Note d'architecture** : clarté, justification des choix, schéma BPMN.

Malus (jusqu'à −3) si la réponse SOAP renvoie des **strings non typées** pour le rapport au lieu d'un ComplexType.

---

11) Bonus (jusqu'à +3)

- (+1) **Caching** simple côté orchestration pour appels CRUD répétés.
- (+1) **Mesures de performance** (latence p95) affichées au shutdown ou sur endpoint santé.
- (+1) **Client web léger** (HTML/JS) qui appelle un **adapter** REST → SOAP (preuve d'interop).

---

12) Indications & astuces

- **Commencez par le XSD**, puis le WSDL : cela force un contrat stable.

- Testez la **WSDL** tôt (import SoapUI) pour éviter les surprises d'interop.
- Générez des **Faults** explicites (code, raison, détail).
- Maintenez la **pureté** des services métier (pas d'IO ni logs verbeux dedans).
- **Tracez** un `correlationId` (par ex. UUID) pour suivre un appel bout-en-bout.
- Gardez la **formule** et la **règle** minimales mais autorisez l'extension (ex. pondérations).