BUT informatique - IUT - Nantes Université
2025-2026
XXXX, Thibaud Laurens, YYYY, ZZZZ

Rapport de

Conception des tests des fonctionnalités d'une API de bibliothèques

SAE401 - Développement d'une application complexe

Module : Qualité de Développement

Dépôt Git : PRIVE POUR PORTFOLAURENS

1) Contexte et objectifs....2
2) Méthodologie de test....2
3) Couverture fonctionnelle....2
3.1 Campagne de test et instrumentation....2
3.2 Résultats globaux....2
3.3 Lecture de la couverture selon les approches R4.02....3
3.4 Interprétation des écarts de couverture....3
4) Non-régression et développement itératif....3
5) Deux types d'analyse sur deux fonctions....4
5.1 Analyse structurelle - `RemoveAllByCli`....4
5.2 Analyse fonctionnelle - `addLivreInto` (table de décision)....6
6) Défauts détectés et corrections (exemples)....8
7) Organisation d'équipe (travail réel en mode étudiant)....9
8) Conclusion et perspectives....9

1) Contexte et objectifs

Dans le cadre de la SAE4 Développement d'une application complexe, partie R4.02, nous avons mis en place une stratégie de test itérative pour garantir :

- la qualité fonctionnelle des services,
- la non-régression après chaque correction,
- la robustesse des endpoints (cas normaux + cas d'erreur),
- une méthode de travail d'équipe reproductible.

L'application est structurée en couches (routes, controllers, DAO, modèles), avec des tests associés dans *Serveur2/service-main/test* et *Serveur2/service-fetch/test*.

2) Méthodologie de test

Nous avons appliqué une approche combinée :

- tests unitaires des modèles, fonctions de contrôleurs et DAO,
- tests des routes de notre API,
- tests des cas d'erreur (validation, exceptions, propagation des statuts),
- tests de non-régression après correction de bugs,
- isolation des dépendances externes via mocks (notamment pour les APIs en ligne).

3) Couverture fonctionnelle

3.1 Campagne de test et instrumentation

Les tests sont lancés avec l'option de couverture active (*--experimental-test-coverage*) dans:

- *Serveur2/service-main*
- *Serveur2/service-fetch*

La commande couvre les tests de contrôleurs, DAO, modèles et routes. Les appels aux APIs externes (Open Library / Google Books) et à nos propres microservices sont mockées dans les tests du fetch pour garantir des résultats déterministes.

3.2 Résultats globaux

Service	Tests	Pass	Fail	Line %	Branch %	Funcs %
Serveur2/service-main	278	278	0	99.42	97.81	99.04
Serveur2/service-fetch	13	13	0	95.56	80.70	93.33

Total : 291 tests exécutés, 291 valides, 0 échec.

3.3 Lecture de la couverture selon les approches R4.02

- Approche fonctionnelle (boîte noire) : les routes et cas métier sont vérifiés selon les entrées/sorties attendues (codes HTTP, messages d'erreur, objets retournés).
- Approche structurelle (boite blanche) : les chemins internes sont vérifiés (conditions, boucles).
- Approche non-régression : chaque correction importante est suivie de tests pour éviter la réapparition du bug.

3.4 Interprétation des écarts de couverture

- *service-main* est très bien couvert sur les couches (*controller*, *dao*, *model*).
- La couverture est plus basse sur certains fichiers: les routes *routes.mjs* car toutes les branches d'erreur ne sont pas accessibles (certains contrôleurs ne peuvent pas lever d'exception), et les fichiers de configuration *index.mjs* et *proxy.mjs* car la même configuration est appliquée à chaque fois (on ne mocke pas de proxy HTTPS).
- À chaque fois, le `branch %` est inférieur au `line %`, ce qui est cohérent avec la présence de branches défensives pour retourner des valeurs par défaut ou lever des exceptions.

En pratique, certaines branches *try/catch* restent non visitées quand les fonctions appelées ne *throw* pas dans le scénario testé. Cela n'invalide pas la qualité globale, mais indique des pistes d'amélioration (tests supplémentaires avec mocks qui forcent les erreurs).

4) Non-régression et développement itératif

Notre cycle itératif suit ce schéma:

1. implémentation d'une fonctionnalité,
2. exécution des tests existants,
3. détection d'un défaut,
4. ajout/ajustement d'un test reproduisant le défaut,
5. correction du code,

6. exécution de la suite pour valider l'absence de régression.

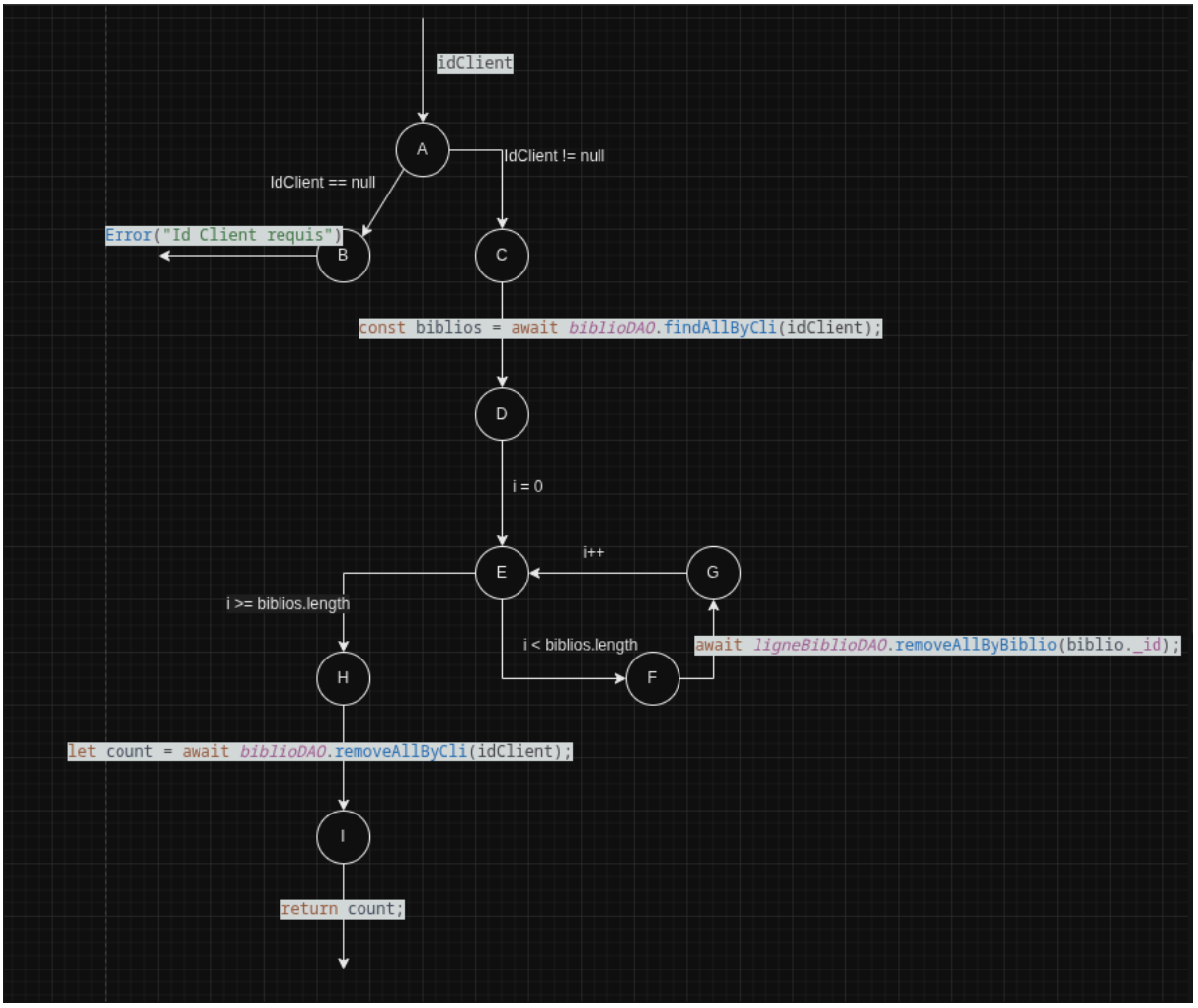
Exemples de points traités pendant les itérations:

- alignement des comportements *findOne* / *findAllByCli* avec les règles métier,
- harmonisation des routes de gestion de bibliothèque,
- correction et mise à jour des tests contrôleurs,
- stabilisation de *fetch100* du *service-fetch* et *livreController.updateBD* du *service-main* via des données mockées.

5) Deux types d'analyse sur deux fonctions

5.1 Analyse structurelle - `RemoveAllByCli`

Graphe associé à la fonction : *RemoveAllByCli*



Légende rapide:

- `A`: entrée
- `B`: validation *idClient* KO (manquant)
- `C`: validation *idClient* OK
- `D`: *findAllByCli*
- `E`: initialisation de la boucle sur les biblios
- `F`: condition de boucle (reste une biblio ?)
- `G`: *removeAllByBiblio* (1 itération)
- `H`: fin de boucle
- `I`: *removeAllByCli* (du DAO) + *return count*

Tableau des routes (1 ligne = 1 route):

Route	**Chemin (graphe/code)**	**Entrée**	**Données de test nécessaires**	**Sortie contrôleur**	**Sortie**
R1	A -> B	`idClient` manquant	Appel avec `idClient = null`	throw `Error("Id Client requis")`, `status=400`	`400`, body `Id Client requis`
R2	A -> C -> D -> E -> F(false) -> H -> I	`idClient` valide, aucune biblio	Client valide sans biblio	retourne `count = 0`	`200`, `{ "message": "0 bibliothèque(s) supprimée(s)", "count": 0 }`
R3	A -> C -> D -> E -> F(true) -> G -> ... -> F(false) -> H -> I	`idClient` valide, biblios présentes	2 biblios + lignes pour client cible, 1 biblio d'un autre client	supprime lignes+biblios du client, retourne `count` (ex `2`)	`200`, `{ "message": "2 bibliothèque(s) supprimée(s)", "count": 2 }`
R4	A -> C -> D -> Exception	erreur DAO sur `findAllByCli`	Mock `findAllByCli` qui throw	exception propagée	`err.status || 500`, body `err.message`

R5	A -> C -> D -> E -> F(true) -> G -> Exception	erreur DAO dans la boucle (`removeAllByBiblio`)	Mock `findAllByCli` non vide + mock `removeAllByBiblio` qui throw	exception propagée, `removeAllByCli` final non execute	500`, body `err.message`
R6	A -> C -> D -> ... -> H -> I -> Exception	erreur DAO sur `removeAllByCli` final	Mock `removeAllByCli` qui throw (liste vide ou non)	exception propagée	`err.status || 500`, body `err.message`

Jeu de données:

- *idClient1* : 2 biblios (`B1`, `B2`) avec lignes
- *idClient2* : 1 biblio témoin (contrôle d'isolation)
- cas validation: *idClient* = *null*
- mocks d'erreur DAO: *findAllByCli*, *removeAllByBiblio*, *removeAllByCli*

5.2 Analyse fonctionnelle - `addLivreInto` (table de décision)

Table de décision pour gérer plusieurs dimensions
Exemple appliqué à *addLivreInto* (*biblioController*)

Fonction analysée:

- Fonction: *biblioCtl.addLivreInto(idClient, isbn, titre)*
- Fichier: *Serveur2/service-main/controller/biblioController.mjs*
- Objectif: ajouter un livre (ISBN) dans une bibliothèque personnelle d'un client.

Règles fonctionnelles observées:

- Si un paramètre manque (*idClient*, *isbn*, *titre*) -> erreur 400.
- Si la bibliothèque n'existe pas pour ce client -> erreur 404.
- Si le livre est déjà présent dans la bibliothèque -> erreur 409.
- Sinon, ajout dans *ligneBiblioDAO* -> succès 200.
- Si un DAO plante -> erreur technique (500).

Variables d'entrée et classes d'équivalence:

Variable	**Type attendu**	**Classes valides**	**Classes invalides**	**Limites intéressantes**

idClient	string non vide	id existant	`null`, `undefined`, `""`	id inexistant
isbn	string non vide	ISBN absent dans la biblio	`null`, `undefined`, `""`, deja present	format numérique non string
titre	string non vide	titre de biblio du client	`null`, `undefined`, `""`	`"  "` (espaces), titre autre client

Table de décision (format proche de l'exemple en image, avec croix):

Entrées / Conditions	**R1**	**R2**	**R3**	**R4**	**R5**	**R6**	**R7**	**R8**
idClient manquant	X							
isbn manquant		X						
titre manquant			X					
biblio inexistante pour ce client				X				
livre déjà présent dans la biblio					X			
erreur DAO lecture (findAllByBiblio)							X	
erreur DAO insertion (add)								X
entrée nominale valide						X		

Sortie attendue	**R1**	**R2**	**R3**	**R4**	**R5**	**R6**	**R7**	**R8**
Error("Url incomplète")	X	X	X					
Error("Bibliothèque non trouvée")				X				
Error("Le livre est déjà dans la bibliothèque")					X			
succès (retour ligne ajoutée)						X		
exception technique propagée							X	X
code HTTP attendu	400	400	400	404	409	200	500	500

Jeux de données de test proposés (en supposant “C1” un *idClient* valide, 9876543210 un *isbn* valide, 9000000001 un *isbn* déjà présent dans la bibliothèque du client, “Wishlist” un *titre* de bibliothèque valide) :

ID test	Entrées (idClient,isbn,titre)	Résultat attendu
T1	null, 9876542310, “Wishlist”	400
T2	“C1”, null, “Wishlist”	400
T3	“C1”, 9876543210, “”	400
T4	“C1”, 8975643210, “Inconnue”	404
T5	“C1”, 9000000001, “Wishlist”	409
T6	“C1”, 9876543210, “Wishlist”	200
T7	lecture DAO en panne	500
T8	Insertion DAO en panne	500

6) Défauts détectés et corrections (exemples)

Défaut observé	Détection par test	Correction appliquée	Validation finale
Comportement incohérent sur recherche unique (`findOne`)	test contrôleur/route en échec	ajustement logique de validation et appels	tests repassent au vert
Retour incomplet sur recherche par client (`findAllByCli`)	test attendu liste vs valeur simple	adaptation du retour pour fournir une liste	tests de régression ajoutés
Dépendance réseau sur `fetch100`	instabilité en cas API externe indisponible	mock `fetch` et jeux de données contrôles	tests déterministes et stables
Codes d'erreur incohérents sur les tests de routes	Les codes d'erreur attendus par les tests n'étaient pas identiques aux codes d'erreur reçus	Ajustement des codes d'erreur renvoyés par le code	tests repassent au vert

7) Organisation d'équipe (travail réel en mode étudiant)

Avec la liste des fonctionnalités à implémenter définies dans nos spécifications et explicitées dans nos diagrammes PBMN ainsi que nos automates, on a fait une liste des fonctionnalités où chacun “réservait” des fonctionnalités au fur et à mesure afin de garder la vision sur qui fait ou a fait quoi. Ainsi, on a tous travaillé sur des fonctionnalités différentes, chacun dans son coin sur une partie du projet. Pour les tests, en revanche, on a essayé d'éviter de tester nos propres fonctionnalités pour avoir un regard plus neutre, bien que ce ne fut pas toujours le cas.

Concrètement, on a fonctionné en petites itérations :

- un membre code une fonctionnalité,
- ce même membre ou un autre membre écrit ou/et complète les tests,
- on corrige ensemble ce qui ne passe pas,

Cette façon de faire nous a aidé à mieux détecter les erreurs et à éviter les oublis.

8) Conclusion et perspectives

La stratégie de test mise en place permet de vérifier méthodiquement la qualité et la non-régression. Les cas limites sont couverts par des routes de test explicites liées au graphe de contrôle. Pour autant, nous aurions pu pousser davantage l’analyse structurelle pour porter notre couverture des branches au maximum. Un autre bémol est de ne pas avoir pris en compte l’analyse de mutation dans nos tests.