RAPPORT D'AUDIT ANGULAR

Évaluation Qualité

wow-character-manager

DATE	ANGULAR	FRAMEWORK DE TEST	FICHIERS ANALYSES
25 février 2026	v20.3.0	Jasmine / Karma	55 TS + 14 HTML

C

67 / 100

01

Synthèse

Le projet **wow-character-manager** est une application Angular 20 bien structurée qui exploite les fonctionnalités modernes du framework : composants standalone, signaux (91 usages), control flow moderne (`@if` , `@for`), injection via `inject()` et lazy loading systématique. L'architecture feature-based avec gestion d'état via `@ngrx/signals` est cohérente et bien organisée.

Cependant, trois problèmes majeurs tirent le score vers le bas : **(1)** la suite de tests est entièrement non fonctionnelle (échec de compilation, 0 tests exécutables), **(2)** l'absence totale de linting (ESLint non configuré) dégrade l'expérience développeur, et **(3)** des fuites mémoire potentielles dans les composants qui souscrivent a des observables long-lived sans nettoyage. Des gains rapides sont possibles en corrigeant les tests, en ajoutant ESLint et en ajoutant `takeUntilDestroyed` aux souscriptions.

02

Scores par categorie

CATÉGORIE	SCORE	PROGRESSION	POIDS	PONDÉRÉ	STATUT
Performance	45%		15%	6.75	Echec
Sécurité	90%		15%	13.50	Bon
Bonnes pratiques Angular	97%		15%	14.55	Bon
Qualité du code	57%		15%	8.55	Echec
DX / Outillage	56%		15%	8.40	Echec
Architecture	75%		15%	11.25	Attention
Tests	40%		10%	4.00	Echec
Global	**67%**		**100%**	**67**	**C**

Bon (≥80%) Attention (60-79%) Echec (<60%) — L'accessibilite est évaluee séparément (voir annexe).

03

Problèmes critiques

Constats a fort impact. A traiter en priorite.

Suite de tests entièrement non fonctionnelle

CRIT.

`*.spec.ts (16 fichiers) — Tests`

Constat : La compilation des tests echoue avant meme l'execution. Plusieurs fichiers spec contiennent des imports invalides, des références a des propriétés/méthodes inexistantes et des mocks incomplets. Aucun des 303 tests déclarés ne peut etre execute.

Impact : Zero couverture de tests en pratique. Les regressions ne sont détectées qu'en production. Aucune garantie de qualité sur les fonctionnalités existantes.

CORRECTION SUGGÉRÉE

Corriger les imports casses dans `character-detail.component.spec.ts` (propriété `server` manquante sur le mock), `character-form.component.spec.ts` (imports relatifs invalides, méthodes inexistantes), `character-list.component.spec.ts` (imports relatifs invalides), `great-vault-calculation.service.spec.ts` (propriétés du modèle inexistantes : `slotsEarned`, `slot1Reward`), `activity.service.spec.ts` (problèmes de widening de type) et `app.spec.ts` (contenu de template attendu incorrect). Valider avec `ng test`.

Fuites mémoire potentielles dans les composants

CRIT.

`character-form.component.ts, settings.component.ts — Performance`

Constat : Des souscriptions à des observables long-lived (`valueChanges`, `statusChanges`) sont créées sans mécanisme de nettoyage. Ni `takeUntilDestroyed`, ni `DestroyRef`, ni `unsubscribe` ne sont utilisés dans ces composants.

Impact : Fuites mémoire progressives, particulièrement lors de navigations repetees. Les callbacks continuent d'etre executes apres la destruction du composant, causant des comportements imprevisibles et une degradation progressive des performances.

CORRECTION SUGGÉRÉE

Injecter `DestroyRef` via `inject(DestroyRef)` et ajouter `takeUntilDestroyed(this.destroyRef)` dans le pipe de chaque souscription long-lived. Exemple : `this.form.valueChanges.pipe(takeUntilDestroyed(this.destroyRef)).subscribe(...)`.

Absence totale d'ESLint

CRIT.

`package.json, racine du projet — DX / Outillage`

Constat : Aucune configuration ESLint n'est présente dans le projet. Pas de fichier `eslint.config.js`, pas de dependance `@angular-eslint`, pas de script `lint` dans `package.json`.

Impact : Les erreurs de style, les imports inutilisés, les violations de conventions Angular et les problèmes potentiels ne sont jamais détectés automatiquement. La qualité du code repose entièrement sur la vigilance humaine.

CORRECTION SUGGÉRÉE

Exécuter `ng add @angular-eslint/schematics` pour configurer ESLint avec les règles Angular. Ajouter un script `"lint": "ng lint"` dans `package.json`.

Usage systemique du type `any` dans le code de production

CRIT.

`~34 occurrences dans 14 fichiers — Qualité du code`

Constat : Le type `any` est utilisé dans environ 34 emplacements du code de production, notamment dans les modèles API (`cached-api-data.model.ts`), les services (`api-cache.service.ts` , `vault-rewards-calculator.service.ts`) et les composants. Certains validateurs utilisent `typeof === 'object'` au lieu de type guards stricts.

Impact : Annulation des benefices de TypeScript strict. Les erreurs de type ne sont détectées qu'a l'execution. Le refactoring devient risque sans filet de sécurité du compilateur.

CORRECTION SUGGÉRÉE

Definir des interfaces typees pour les reponses API. Remplacer `any` par `unknown` suivi de type guards, ou par des types génériques. Priorite : les fichiers modèles et les services de cache.

Couverture de tests insuffisante (29% par fichier)

CRIT.

`src/app/ — Tests`

Constat : Seuls 16 fichiers spec existent pour 55 fichiers source TypeScript, soit un taux de couverture par fichier de 29%. De nombreux services critiques (`blizzard-api.service.ts` , `api-cache.service.ts` , `weekly-progress.service.ts`) n'ont pas de tests dedies.

Impact : Les services métier essentiels ne sont pas proteges par des tests. Les modifications dans ces fichiers peuvent introduire des regressions silencieuses.

CORRECTION SUGGÉRÉE

Prioriser la creation de tests pour les services métier critiques : `blizzard-api.service.ts` , `api-cache.service.ts` , `weekly-progress.service.ts` , `vault-rewards-calculator.service.ts` . Viser 60% de couverture minimale a court terme.

04

Avertissements

Points a corriger — non bloquants, mais impactants.

4 composants sans OnPush

WARN

`settings.component.ts, vault-slot.component.ts, app-toolbar.component.ts, app.ts — Performance`

10 composants sur 14 utilisent `OnPush` . Les 4 restants déclenchent des cycles de détection de changement complets sur chaque événement. Ajouter `changeDetection: ChangeDetectionStrategy.OnPush` pour homogénéiser la stratégie.

6 souscriptions manuelles dans les composants

WARN

`character-form.component.ts, app-toolbar.component.ts, settings.component.ts, vault-progress-card.component.ts — Performance`

Des appels `.subscribe()` directs dans les composants au lieu d'utiliser le pipe `async` ou des signaux. Cela complique la gestion du cyclé de vie et peut mener a des fuites mémoire si les souscriptions ne sont pas nettoyees.

Pas de stratégie de preloading configurée

WARN

`app.config.ts — Performance`

Les routes sont lazy-loaded mais aucune stratégie de preloading n'est definie. L'ajout de `withPreloading(PreloadAllModules)` dans `provideRouter()` permettrait de pre-charger les modules en arriere-plan apres le chargement initial.

34 instructions console.log/debug/info dans le code de production

WARN

`vault-rewards-calculator.service.ts (8), character-refresh.service.ts (4), store/*.ts (11), etc. — Sécurité / Qualité`

34 instructions de logging reparties dans 11 fichiers source. Ces sorties console exposent des details internes de l'application et polluent la console du navigateur en production. Utiliser un service de logging avec des niveaux configurables ou supprimer ces instructions.

47 imports relatifs profonds (4+ niveaux)

WARN

`Multiples fichiers dans src/app/ — Architecture`

47 imports utilisent 4 niveaux ou plus de `../`. Aucun alias de chemin (`paths`) n'est configuré dans `tsconfig.json`. Ces imports sont fragiles, difficiles a lire, et cassent facilement lors de restructurations.

Services dupliqués : WeeklyResetService vs ResetService

WARN

`weekly-reset.service.ts, reset.service.ts — Architecture / Qualité`

Deux services gerent la reinitialisation hebdomadaire avec des hypotheses différentes : `WeeklyResetService` utilisé mercredi 15:00 UTC tandis que `ResetService` utilise mercredi 10:00 heure locale. Ce conflit peut produire des resultats incoherents selon le service utilise.

Constante CLASS_COLORS dupliquée dans 3+ fichiers

WARN

`Multiples fichiers — Qualité du code`

La constante `CLASS_COLORS` est definie dans au moins 3 emplacements avec des valeurs hexadecimales différentes. Cela provoque des incoherences visuelles et complique la maintenance. Centraliser dans `shared/constants/`.

Absence de fichiers d'environnement

WARN

`src/environments/ — DX / Outillage`

Aucun fichier `environment.ts` ou `environment.prod.ts` n'a été trouvé. La configuration d'environnement est absente, ce qui rend difficile la gestion des URLs d'API, des clés de configuration, et la distinction dev/production.

Souscription vide dans blizzard-api.service.ts

WARN

`blizzard-api.service.ts:645 — Qualité du code`

Un appel `.subscribe()` sans arguments ni callbacks a la ligne 645 du service API Blizzard. Les erreurs de cette souscription sont silencieusement ignorées. Ajouter au minimum un handler `error` ou utiliser `catchError` dans le pipe.

Manipulation directe du DOM dans theme.service.ts

WARN

`theme.service.ts — Architecture`

Accès direct au DOM via `document.querySelector` ou equivalent dans le service de theme. Cela peut causer des problèmes avec le SSR et contourne l'abstraction de rendu Angular. Envisager l'utilisation de `Renderer2` ou de `inject(DOCUMENT)`.

05

Notes informatives

Code mort détecté : La classe `CharacterFormGroup` dans `forms/character-form-group.ts` est entièrement implémentée mais jamais utilisée. Elle a été remplacée par `SimpleCharacterFormGroup`. La méthode `getDetailedVaultRewards()` dans `vault-rewards-calculator.service.ts` est également marquée comme deprecated et inutilisée.

Validateurs faibles dans api-cache.service.ts : Des vérifications `typeof === 'object'` sont utilisées pour valider les données en cache au lieu de type guards stricts.

skipLibCheck active : L'option `skipLibCheck: true` dans `tsconfig.json` masque les erreurs de type dans les dépendances tierces. Désactiver si possible pour une meilleure sécurité de type.

withInterceptorsFromDi utilisé : L'application utilise `withInterceptorsFromDi()` au lieu de l'API fonctionnelle `withInterceptors()`. Migrer vers les intercepteurs fonctionnels pour suivre les pratiques modernes Angular.

Pas de hooks pre-commit : Ni `husky` ni `lint-staged` ne sont configurés. L'ajout de hooks pre-commit empêcherait les commits de code non conforme.

Formulaires bien types (point positif) : Les classes `SimpleCharacterFormGroup` et `CharacterFormGroup` étendent `FormGroup` avec des types génériques. C'est une bonne pratique rarement vue dans les projets Angular.

Aucune API signal input/output utilisée : Le projet n'utilisé ni `@Input()` / `@Output()` classiques ni les nouvelles API `input()` / `output()`. La communication se fait principalement via les stores de signaux.

Gestion d'erreur robuste (point positif) : 62 occurrences de `catchError` / `catch` détectées dans les services, indiquant une bonne stratégie de gestion d'erreur HTTP.

06

Plan d'action

Gains rapides

Corrections simples a fort impact. Commencez par là.

1. **Corriger les fichiers spec casses** — Restaurer une suite de tests fonctionnelle (+15 pts Tests). Mettre a jour les imports, mocks et propriétés références dans les 6 fichiers spec en erreur.
2. **Ajouter ESLint avec @angular-eslint** — Améliorer la qualité et l'outillage (+15 pts DX). Une seule commande : `ng add @angular-eslint/schematics`.
3. **Ajouter takeUntilDestroyed() aux souscriptions** — Éliminer les fuites mémoire (+10 pts Performance). Concerne 2 composants : `character-form` et `settings`.
4. **Ajouter OnPush aux 4 composants manquants** — Homogénéiser la détection de changement (+5 pts Performance).
5. **Supprimer les console.log** — Nettoyer la sortie de production (+3 pts Sécurité, +3 pts Qualité). 34 occurrences dans 11 fichiers.

Feuille de route

COURT TERME · < 1 SEMAINE

- Corriger les 6 fichiers spec en erreur de compilation
- Ajouter `takeUntilDestroyed()` dans les composants avec souscriptions long-lived
- Ajouter `OnPush` aux 4 composants manquants
- Supprimer les 34 `console.log` du code de production
- Supprimer le code mort (`CharacterFormGroup`, méthode deprecated)

MOYEN TERME · 1-4 SEMAINES

- Installer et configurer ESLint avec `@angular-eslint`
- Configurer des alias de chemin (`@app/*`, `@shared/*`) dans tsconfig
- Unifier les services de reset (WeeklyResetService / ResetService) en un seul
- Centraliser la constante `CLASS_COLORS` dans `shared/constants/`
- Remplacer les ~34 usages de `any` par des types stricts
- Créer les fichiers d'environnement (dev, prod)
- Ajouter `withPreloading(PreloadAllModules)` au routeur

LONG TERME · 1+ MOIS

- Augmenter la couverture de tests a 60% + (priorite : services métier)
- Configurer les hooks pre-commit (husky + lint-staged)
- Migrer vers les intercepteurs fonctionnels (`withInterceptors`)
- Considérer une **Clean Architecture** : le projet a une logique métier non triviale (gestion de coffre-fort, calculs de recompenses, progression hebdomadaire) actuellement couplée a l'infrastructure Angular. Séparer la logique domaine du framework améliorerait la testabilité et la maintenabilité. Le skill `angular-clean-arch-scaffold` peut aider a restructurer le projet.

07

Résultats détaillés

#	FICHIER	CAT.	SEV.	PROBLÈME	SUGGESTION
1	`*.spec.ts (16 fichiers)`	Tests	CRIT.	Suite de tests non fonctionnelle : échec de compilation, 0 tests exécutables	Corriger imports, mocks et références dans les fichiers spec
2	`character-form.component.ts`	Performance	CRIT.	Souscription a `valueChanges` sans nettoyage (fuite mémoire)	Ajouter `takeUntilDestroyed()`
3	`settings.component.ts`	Performance	CRIT.	2 souscriptions a `valueChanges` / `statusChanges` sans nettoyage	Ajouter `takeUntilDestroyed()`
4	`package.json, racine`	DX / Outillage	CRIT.	ESLint non configuré, aucune règle de linting	`ng add @angular-eslint/schematics`
5	`~14 fichiers production`	Qualité	CRIT.	~34 usages de `any` dans le code de production	Remplacer par des types stricts, `unknown` + type guards
6	`src/app/ (16/55 fichiers)`	Tests	CRIT.	Couverture de test a 29% par fichier, services critiques non testes	Ajouter des tests pour les services métier prioritaires
7	`settings, vault-slot, app-toolbar, app`	Performance	WARN	4 composants sans `ChangeDetectionStrategy.OnPush`	Ajouter `changeDetection: ChangeDetectionStrategy.OnPush`
8	`4 fichiers composant`	Performance	WARN	6 souscriptions manuelles `.subscribe()` dans les composants	Préférer le pipe `async` ou convertir en signaux
9	`app.config.ts`	Performance	WARN	Pas de stratégie de preloading configurée	Ajouter `withPreloading(PreloadAllModules)`
10	`11 fichiers source`	Sécurité / Qualité	WARN	34 instructions `console.log/debug/info` en production	Supprimer ou utiliser un service de logging avec niveaux
11	`Multiples fichiers`	Architecture	WARN	47 imports relatifs profonds (4+ niveaux `../`)	Configurer `paths` dans `tsconfig.json`
12	`weekly-reset.service.ts, reset.service.ts`	Architecture	WARN	Services dupliqués avec des heures de reset différentes	Fusionner en un seul service avec l'heure UTC correcte
13	`3+ fichiers`	Qualité	WARN	Constante `CLASS_COLORS` dupliquée avec valeurs différentes	Centraliser dans `shared/constants/class-colors.ts`
14	`src/environments/`	DX / Outillage	WARN	Pas de fichiers d'environnement configurés	Créer `environment.ts` et `environment.prod.ts`

#	FICHIER	CAT.	SEV.	PROBLÈME	SUGGESTION
15	`blizzard-api.service.ts:645`	Qualité	WARN	Souscription vide `.subscribe()` sans gestion d'erreur	Ajouter un handler `error` ou `catchError`
16	`theme.service.ts`	Architecture	WARN	Manipulation directe du DOM	Utiliser `Renderer2` ou `inject(DOCUMENT)`
17	`forms/character-form-group.ts`	Qualité	INFO	Code mort : classe `CharacterFormGroup` jamais utilisée	Supprimer le fichier (l'historique git preserve le code)
18	`vault-rewards-calculator.service.ts`	Qualité	INFO	Méthode `getDetailedVaultRewards()` deprecated et inutilisée	Supprimer la méthode deprecated
19	`tsconfig.json`	DX / Outillage	INFO	`skipLibCheck: true` active	Désactiver si les dépendances le permettent
20	`app.config.ts`	Angular	INFO	`withInterceptorsFromDi()` au lieu de `withInterceptors()`	Migrer vers les intercepteurs fonctionnels
21	`package.json`	DX / Outillage	INFO	Pas de hooks pre-commit (husky / lint-staged)	Installer `husky` et `lint-staged`

ANNEXE

Rapport d'accessibilite

Cette section évalue l'accessibilite séparément et n'affecte pas le score principal. Pertinent pour les applications publiques et la conformité règlementaire (RGAA, WCAG 2.1).

Résumé

L'accessibilite du projet est **insuffisante**. Aucun attribut `aria-*` ni `role` n'est utilisé dans les templates. Les 8 images présentes n'ont pas d'attribut `alt`. Aucune stratégie de navigation au clavier ou d'annonce de changement de route n'a été détectée. Ce point est particulièrement important si l'application est destinée a un usage public.

#	FICHIER	CHECK	SEV.	PROBLÈME	SUGGESTION
1	`Templates HTML (8 images)`	A11Y-01	WARN	8 balises `<img>` sans attribut `alt`	Ajouter `alt="description"` ou `alt=""` pour les images décoratives
2	`Templates HTML`	A11Y-13/14	WARN	Aucun attribut `aria-*` ni `role` dans les templates	Ajouter des attributs ARIA aux éléments interactifs et `aria-live` aux zones dynamiques
3	`app.routes.ts`	A11Y-17	WARN	Aucune stratégie de titre de route pour les lecteurs d'écran	Ajouter la propriété `title` aux routes ou configurer un `TitleStrategy`

ANNEXE

Grille d'évaluation détaillée

Chaque catégorie est évaluée sur une grille de points de contrôle. La gomette indique le résultat de l'audit pour ce projet. ● OK ● À améliorer ● Problème détecté ● Non applicable

Performance

60 / 100

●	**Lazy loading des routes**	Réduit le bundle initial et accélère le premier affichage en ne chargeant que le code nécessaire.
●	**Stratégie de préchargement**	Sans préchargement, les modules lazy sont chargés à la demande, causant des latences de navigation.
●	**Composants en OnPush**	Sans OnPush, Angular recalcule tout le template à chaque événement, même si les données n'ont pas changé.
●	**Pas de déclencheurs zone inutiles**	setTimeout/setInterval dans les composants provoquent des cycles de détection inutiles.
●	**Expressions track dans les boucles**	Sans track, Angular détruit et recrée tous les éléments DOM à chaque changement de liste.
●	**Pas d'appels de fonction impurs dans les templates**	Les méthodes appelées dans les templates sont recalculées à chaque cycle de détection.
●	**Pas de Moment.js**	Moment.js est une librairie lourde et obsolète qui empêche le tree-shaking.
●	**Pipe async ou signaux préférés**	Les .subscribe() manuels dans les composants créent du code impératif et des risques de fuites mémoire.
●	**Pas de souscriptions imbriquées**	Les subscribes imbriqués créent du code spaghetti impossible à maintenir et à gérer en erreur.
●	**Nettoyage des souscriptions**	Les souscriptions non nettoyées provoquent des fuites mémoire progressives en production.

Sécurité

90 / 100

●	**Pas de binding innerHTML**	innerHTML contourne le compilateur de templates Angular et peut ouvrir des vecteurs XSS.
●	**Pas de bypassSecurityTrust**	Désactive complètement le sanitizer Angular — chaque usage doit être justifié.
●	**Pas d'eval()**	eval() exécute du code arbitraire — aucun cas d'usage légitime dans une app Angular.
●	**Pas de document.write**	document.write casse le DOM dans les SPA et peut introduire des vulnérabilités XSS.
●	**Pas de secrets en dur**	Les mots de passe et clés API dans le code source sont exposés à quiconque a accès au repo.
●	**Pas de fichiers .env commités**	Les fichiers .env contiennent souvent des secrets qui ne doivent pas être versionnés.
●	**Fichiers d'environnement sûrs**	Les fichiers environment.ts ne doivent contenir que de la configuration, pas des credentials.
●	**Pas de dépendances vulnérables**	Les dépendances avec des CVE connues exposent l'application à des attaques connues.
●	**Version Angular LTS**	Une version hors LTS ne reçoit plus de correctifs de sécurité ni de bugs.
●	**Pas d'URL HTTP en production**	Les connexions HTTP non chiffrées exposent les données en transit aux interceptions.
●	**HttpClient utilisé pour les appels API**	HttpClient supporte les intercepteurs pour centraliser l'auth, le logging et la gestion d'erreurs.
●	**Pas de console.log en production**	Les console.log peuvent exposer des données sensibles dans la console du navigateur en production.
●	**Pas d'appels alert()**	alert/confirm/prompt bloquent le thread UI et offrent une expérience utilisateur dégradée.
●	**Pas d'instructions debugger**	Les instructions debugger oubliées dans le code gèlent l'application pour tout utilisateur avec les devtools ouvertes.

Bonnes pratiques Angular

75 / 100

●	**Composants standalone**	Les composants standalone simplifient les imports et éliminent le besoin de NgModules.
●	**Pas de barrel files excessifs**	Les barrel files (index.ts) excessifs cassent le tree-shaking et favorisent les dépendances circulaires.
●	**Composants petits et focalisés**	Les composants volumineux mélangent responsabilités UI et logique métier, rendant le code difficile à tester.
●	**Syntaxe de contrôle moderne**	@if/@for/@switch offrent de meilleures performances et une syntaxe plus lisible que *ngIf/*ngFor.
●	**Pas d'APIs dépréciées**	Les APIs dépréciées seront supprimées dans les futures versions, créant une dette technique.
●	**Hooks de cycle de vie modernes**	afterNextRender/afterRender sont plus adaptés pour les manipulations DOM que ngAfterViewInit.
●	**Signaux pour l'état des composants**	Les signaux offrent une réactivité fine-grained sans les complexités de RxJS pour l'état local.
●	**Inputs/outputs signal**	Les input()/output() signal offrent un meilleur typage et une intégration native avec le système de signaux.
●	**Requêtes signal viewChild/ contentChild**	Les requêtes signal s'intègrent au graphe de réactivité et éliminent les problèmes de timing.
●	**Fonction inject()**	inject() simplifie l'injection et permet l'utilisation en dehors des constructeurs (guards, interceptors).
●	**Portée des services appropriée**	Un scope mal choisi cause des singletons inattendus ou des instances dupliquées.
●	**provideRouter utilisé**	provideRouter() est l'API fonctionnelle moderne, plus légère que RouterModule.forRoot().
●	**Guards et resolvers fonctionnels**	Les guards fonctionnels sont plus simples, composables et tree-shakeable que les class-based.
●	**Formulaires fortement typés**	Les formulaires non typés font perdre tout le bénéfice du typage TypeScript sur les valeurs du formulaire.
●	**Validation des formulaires cohérente**	Une validation incohérente laisse passer des données invalides et dégrade l'expérience utilisateur.
●	**Intercepteurs fonctionnels HttpClient**	withInterceptors() est l'API moderne, plus composable et tree-shakeable que les class-based.
●	**Gestion d'erreurs HTTP**	Sans gestion d'erreurs, les échecs réseau laissent l'utilisateur sans feedback et l'app dans un état incohérent.
●	**Pas d'appels HTTP dans les composants**	Les appels HTTP directs dans les composants violent la séparation des préoccupations et rendent le code intestable.

Qualité du code

55 / 100

●	**Pas de type any**	Le type any désactive la vérification de type et propage des erreurs silencieuses dans tout le code.
●	**Pas d'assertions de type injustifiées**	Les assertions 'as' contournent le vérificateur de types et masquent des bugs potentiels.
●	**Interfaces pour les shapes de données**	Les interfaces donnent de meilleurs messages d'erreur et sont plus performantes que les type aliases pour les objets.
●	**Pas de blocs catch vides**	Les catch vides avalent silencieusement les erreurs, rendant le debugging en production impossible.
●	**Typage des erreurs dans les catch**	Sans typage explicit, les erreurs catch sont implicitement 'any', perdant toute sécurité de type.
●	**Gestion d'erreurs dans les observables**	Un subscribe sans gestion d'erreur crashe silencieusement, laissant l'UI dans un état incohérent.
●	**Gestion d'erreurs HTTP cohérente**	Sans stratégie cohérente, chaque développeur gère les erreurs différemment, créant une expérience utilisateur incohérente.
●	**Pas de console.log en production**	Les console.log polluent la sortie, peuvent exposer des données sensibles et indiquent du code de debug oublié.
●	**Pas de marqueurs TODO/FIXME**	Les TODO dans le code sont de la dette technique invisible — ils doivent être trackés dans un issue tracker.
●	**Code commenté détecté**	Le code commenté crée de la confusion et du bruit — le git history conserve l'historique.
●	**Formatage cohérent du code**	Sans formateur configuré, le style de code diverge entre développeurs et pollue les diffs git.
●	**Fichiers en kebab-case**	Le kebab-case est la convention Angular standard et assure la cohérence des noms de fichiers.
●	**Convention de nommage cohérente**	Mixer anciennes et nouvelles conventions de nommage dans un même projet crée de la confusion.
●	**Utilisation cohérente des préfixes**	Les préfixes de sélecteur identifient l'origine d'un composant et évitent les collisions de noms.
●	**Pas de callbacks profondément imbriqués**	Les callbacks imbriqués (pyramid of doom) sont illisibles et impossibles à débugger.
●	**Fonctions de taille raisonnable**	Les fonctions de plus de 50 lignes sont difficiles à comprendre, tester et maintenir.
●	**Pas de god services**	Un service avec trop de responsabilités viole le principe de responsabilité unique et devient intestable.
●	**Pas d'imports inutilisés**	Les imports inutilisés encombrent le code et masquent les vraies dépendances.
●	**Pas de code inatteignable**	Le code après return/throw/break ne s'exécute jamais et crée de la confusion.

DX / Outillage

40 / 100

●	**Mode strict TypeScript activé**	Le mode strict détecte des classes entières de bugs à la compilation plutôt qu'en production.
●	**Options strictes du compilateur Angular**	strictTemplates détecte les erreurs de type dans les templates au moment de la compilation.
●	**Alias de chemins configurés**	Sans alias, les imports relatifs profonds (../../..) rendent le code illisible et fragile au refactoring.
●	**Pas de skipLibCheck**	skipLibCheck masque des erreurs de type dans les dépendances tierces, réduisant la fiabilité.
●	**ESLint configuré**	Sans linter, les problèmes de qualité et les anti-patterns ne sont jamais détectés automatiquement.
●	**Règles Angular ESLint activées**	@angular-eslint fournit des règles spécifiques qui détectent les anti-patterns Angular courants.
●	**Script lint dans package.json**	Sans script lint, les développeurs n'ont pas de moyen standard de vérifier la qualité du code.
●	**Formateur de code configuré**	Sans formateur, chaque développeur utilise un style différent, polluant les diffs et les code reviews.
●	**EditorConfig présent**	EditorConfig assure des paramètres d'éditeur cohérents (indentation, encodage) entre les IDE.
●	**angular.json bien configuré**	Une configuration build/serve/test incomplète bloque le workflow de développement.
●	**Source maps pour le développement**	Sans source maps, le debugging se fait dans du code minifié/bundlé — une perte de temps énorme.
●	**Environnements séparés**	Sans séparation dev/prod, les configurations de développement fuient en production (ou inversement).
●	**Scripts npm essentiels définis**	Des scripts standardisés (start, build, test, lint) permettent à tout nouveau développeur de démarrer immédiatement.
●	**Hooks pre-commit configurés**	Sans hooks, du code non conforme entre dans le repo et est découvert trop tard en code review ou en CI.
●	**Version Angular CLI à jour**	Un CLI en retard empêche l'accès aux dernières optimisations, schémas et commandes.
●	**Valeurs par défaut des schematics**	Sans defaults (changeDetection, style), chaque ng generate produit du code incohérent avec le reste du projet.

Architecture

75 / 100

Structure de dossiers cohérente	Une structure incohérente force chaque développeur à "deviner" où trouver ou placer du code.
Dossiers feature auto-contenus	Des features éclatées dans l'arborescence rendent impossible de comprendre le périmètre d'une fonctionnalité.
Code partagé dans des dossiers dédiés	Du code dupliqué entre features multiplie les bugs et l'effort de maintenance.
Structure plate dans les features	Un nesting excessif dans les features rend la navigation difficile et signale une mauvaise séparation.
Composants sans logique métier	La logique métier dans les composants les rend intestables et couplés à l'UI.
Services à responsabilité unique	Un service fourre-tout est impossible à mocker, tester et faire évoluer indépendamment.
Templates sans logique complexe	Des expressions complexes dans les templates sont recalculées à chaque cycle et rendent le HTML illisible.
Stratégie de styles cohérente	Mixer ViewEncapsulation.None et le défaut crée des effets de bord CSS imprévisibles.
Injection de dépendances cohérente	Instancier des services avec new contourne le DI, rendant les tests et le remplacement impossibles.
Providers organisés	Des providers éparpillés dans plein de composants rendent la configuration difficile à tracer.
Pas de dépendances circulaires	Les dépendances circulaires causent des erreurs d'injection au runtime et des bugs subtils.
Features lazy-loadées	Sans lazy loading, tout le code est chargé au démarrage, augmentant le temps de chargement initial.
Pas d'imports cross-feature profonds	Importer les fichiers internes d'une autre feature crée un couplage fort qui bloque le refactoring.
Frontières API claires	Sans API claire entre features, tout devient interconnecté et chaque changement a des effets de bord.
Pas d'imports relatifs profonds	Les imports ../../../../ sont fragiles, illisibles et cassent au moindre déplacement de fichier.
Ordre des imports cohérent	Un ordre d'imports incohérent pollue les diffs git et rend les imports plus durs à scanner visuellement.
Gestion d'état cohérente	Mixer plusieurs patterns de state management crée de la confusion sur où trouver et modifier l'état.
État partagé dans des services/stores	L'état partagé dans des propriétés de composants crée des sources de vérité multiples et des bugs de synchronisation.

	Pas de dépendances store-to-store	Les stores qui s'appellent entre eux créent un flux de données impossible à tracer.
●	**Pas de pattern event bus global**	Un Subject générique utilisé comme bus d'événements crée un flux spaghetti — personne ne sait qui déclenche quoi.
●	**Pas de manipulation directe du DOM**	document.querySelector contourne l'abstraction de rendu Angular, casse le SSR et crée des bugs subtils.
●	**Pas d'état mutable partagé**	Des objets/arrays mutables exposés publiquement permettent des modifications incontrôlées qui échappent au change detection.
●	**Patterns async cohérents**	Mixer Promises et Observables pour le même type d'opération rend la composition et la gestion d'erreurs incohérentes.

Tests

10 / 100

●	**Couverture de code**	La couverture mesure quelle proportion du code est exercée par les tests — sans elle, les régressions passent inaperçues.
●	**Chemins critiques testés**	Les services et composants clés sans tests sont des bombes à retardement — un changement peut tout casser sans alerte.
●	**Configuration de tests valide**	Sans configuration valide, aucun test ne peut s'exécuter et la couverture est fictive.
●	**Tests suivent le pattern AAA**	Le pattern Arrange-Act-Assert structure les tests de manière lisible et maintenable.
●	**Descriptions de tests significatives**	Des descriptions vagues rendent impossible de comprendre ce qu'un test vérifie quand il échoue.
●	**Pas de tests focused ou skipped**	fdescribe/fit font tourner un seul test en CI. xdescribe/xit masquent des échecs. Les deux sont dangereux.
●	**Tests organisés de manière cohérente**	Des tests éparpillés ou mal rangés sont plus durs à trouver, maintenir et exécuter sélectivement.
●	**Tests avec assertions**	Un test sans expect ne vérifie rien — il passe toujours, donnant une fausse impression de qualité.
●	**Pas de tests qui vérifient seulement les spies**	Un test qui vérifie uniquement qu'un spy a été appelé teste l'implémentation, pas le comportement.
●	**Tests ne testent pas les détails d'implémentation**	Accéder aux membres privés dans les tests les rend fragiles — tout refactoring les casse.
●	**Pas de valeurs magiques dans les tests**	Des valeurs en dur sans contexte rendent les tests cryptiques et difficiles à maintenir.
●	**Utilisation appropriée de TestBed**	TestBed pour de la logique pure (services sans DI) ralentit les tests inutilement.
●	**Mocking correctement effectué**	Des mocks artisanaux incomplets laissent passer des bugs et sont plus durs à maintenir.
●	**Tests async gérés correctement**	Les setTimeout dans les tests créent des tests non-déterministes qui échouent de manière intermittente.

Louis-Jean Claeyssen