Document d’architecture technique

Sommaire

Dernière modification : 2023-06-12

Etat : *En cours*

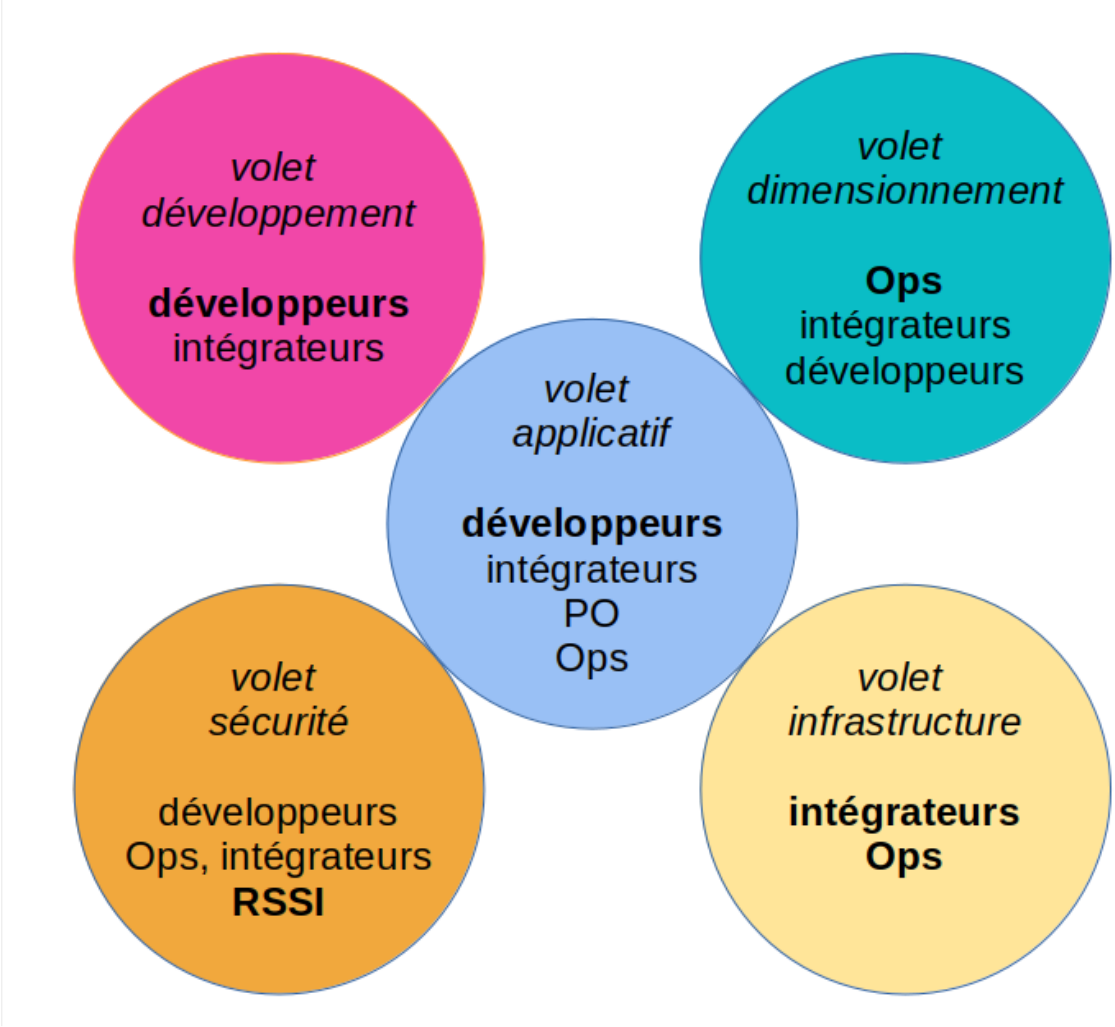
Ce dossier est constitué :

- d'un volet applicatif présentant le contexte général et l'architecture applicative ;
- d'un volet développement présentant l'architecture logicielle ;
- d'un volet infrastructure présentant les aspects d'architecture techniques ou physiques ;
- d'un volet dimensionnement présentant les aspects liés aux performances et aux sollicitations ;
- d'un volet sécurité.

Chaque volet expose les contraintes, exigences puis solutions mises en œuvre.

Le glossaire du projet est disponible ici.

À lire en priorité en fonction de votre rôle :



Ce dossier d'architecture a été réalisé à partir de ce modèle.

1. Volet architecture applicative

Les autres volets du dossier sont accessibles d'ici.

Cette section décrit les modules applicatifs en jeu et leurs échanges.

1.1. Documentation de référence

Table 1. Références documentaires

N°	Version	Titre/URL du document	Détail
1	3.2	https://docs.google.com/document/d/1pMPZekT_XMff5s5Jra1vCpLX1I4NqYWR	La proposition commerciale
2	0.3	https://docs.google.com/document/d/1WuxDCi0bDFyBTOuSaX1i-qgJR6kGl2_lkQ2u3sNbmkU/edit	Les spécifications générales

1.2. Contexte général

Dans le cadre d'une stratégie globale d'amélioration continue, la société ECOMICRO souhaite s'inscrire dans une dynamique de modernisation et d'optimisation de son système d'information.

1.2.1. Objectifs

L'objectif est de réaliser un outil qui réponde aux besoins exprimés par ECOMICRO, en se concentrant sur les points de fonction essentiels, qui soit rapide à mettre en place et à prendre en main, dans un budget cohérent, puis de l'améliorer.

L'informatisation des processus actuellement réalisés à la main sera un gain notable de productivité pour ECOMICRO et permettra de pérenniser l'accès aux informations.

1.2.2. Existant

La gestion globale de ECOMICRO est aujourd'hui supportée par le biais de divers supports digitaux et papiers. La plupart de ces systèmes ne sont pas connectés entre eux de manière automatique par le biais d'interfaçages optimisés.

La société EcoMicro ne possède pas à l'heure actuelle de logiciel de gestion. L'intégralité des éléments documentaire en lien avec le client (devis, commande, facture, planning, exploitation des déchets, reconditionné, revente) est gérée via des outils papier ou Microsoft.

Différents fichiers Excel sont également utilisés et tenus à jour pour la gestion quotidienne des différents services.

Des documents Word sont réalisés afin d'effectuer de la prise d'information et du rédactionnel, l'usage de Word et d'au moins un PDF sont bel et bien réalisés par l'entreprise. Cet usage concerne les devis envoyés au client pour ce qui est du PDF. Le bordereau de suivi des déchets, la procédure d'audit UC, ainsi que les problèmes récurrents avant ticket SAV sont réalisés sur word.

Enfin, des documents sont imprimés et tenus à jour manuellement par le personnel, principalement au niveau de l'atelier de production.

1.2.3. Acteurs

1.2.3.1. Acteurs internes

Table 2. Liste des acteurs internes

Acteur	**Population**
Direction	1
Commerciaux	2
ADV	1
Responsable de l'exploitation	1
Opérateurs d'exploitation	2

Acteur	Population
Responsables du reconditionnement	2
Gestionnaire administrative	1

1.2.3.2. Acteurs externes

Table 3. Liste acteurs externes

Acteur	Population
Utilisateurs non connectés	N/A
Client	N/A

1.3. Contraintes

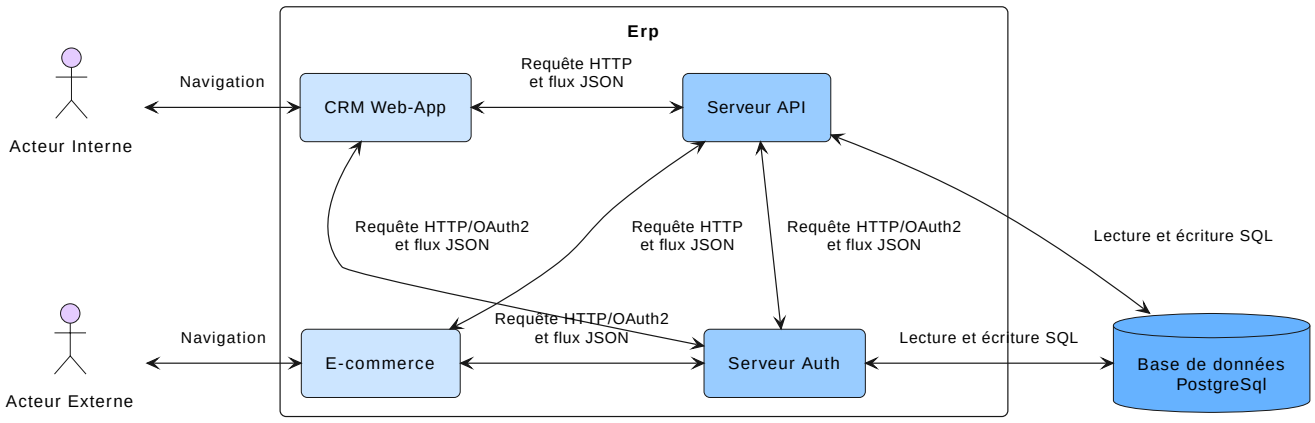
Aucune contraintes n'a été recensée.

1.4. Exigences

Le concept MVP (Minimum Viable Product) doit orienter le développement du project en lot 1.

1.5. Architecture

L'ERP d'EcoMicro est composée de deux applications webs, un CRM pour les acteurs internes, et un site e-commerces pour les acteurs externes.



Il s'agit de deux front-end distincts exposés aux utilisateurs qui appels les web service REST exposé par un même back-end.

Les données reçues et envoyées par les web service seront au format JSON, avec des noms de propriétés écrits en camelCase, sauf cas particulier de web service renvoyant des fichiers (par exemple csv).

1.5.1. Général

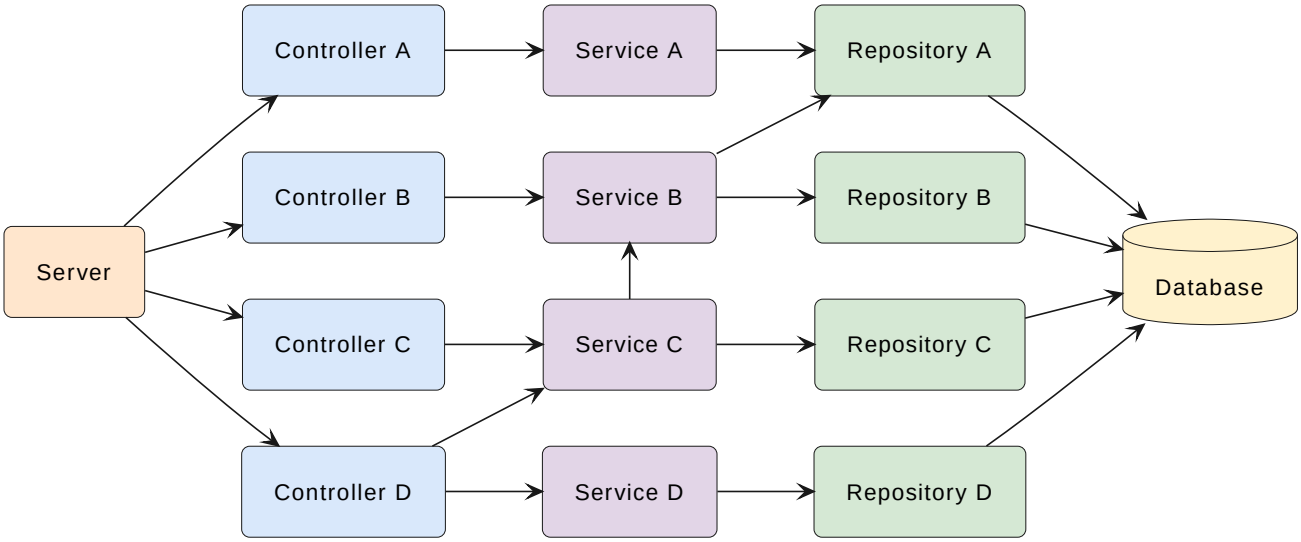
Pour la partie front-end, EcoMicro utilise le framework Angular, avec le langage TypeScript.

Il y a deux project applicatif, un pour le CRM et un pour le site e-commerce.

Et des librairies sont mises en place pour le code commun entre les deux.

Chaque module fonctionnel est en lazy-loading, ce qui permet de charger les modules en fonction du besoin.

Pour la partie back-end, EcoMicro utilise le framework Spring Boot et utilise une architecture en couches (contrôleur, service, repository).



L'architecture en layer permet une maintenance plus facile et une séparation des fonctionnalités, entrainant une meilleure réutilisation des composants.

1.5.2. La partie Front-end détaillée

L'application est composée de plusieurs couches :

- Les composants, ils forment l'IHM, et sont séparés en deux catégories : les composants réutilisables, et les pages. Ils ont pour objectif :
  - La récupération des données venant des services.
  - L'affichage des informations utiles pour l'utilisateur.
- Les pages, ils forment l'IHM. Ils ont pour objectif :
  - La récupération des données venant des services.
  - L'affichage des informations utiles pour l'utilisateur.
- Les services, ils contiennent le code métier de l'application. Ils ont pour objectif :
  - Une meilleure isolation du code réutilisable.
  - Une séparation des responsabilités.
  - Une implémentation simplifiée des tests unitaire sur le code métier.
- Les resources, ils ont pour vocation à effectuer les appels *HTTP* permettant de récupérer les objects en provenance du back-end. Ils ont pour objectif :
  - Une meilleure séparation entre la récupération des données et leurs traitements par la

couche service

- Une implémentation simplifiée des tests unitaire par la création de mock.

1.5.2.1. Règles générales

- Les logs d'erreurs seront remontés au back pour être inscrits dans la stack ELK, via le `ErrorHandler` d'Angular (pas mis en place lors du MVP).
- L'utilisation des interfaces est à privilégier pour la description d'objet sans logique et non métier.
- L'utilisation des énumérations constantes est à privilégier pour le stockage des constantes.
- Les énumérations non-constantes seront à utiliser pour l'utilisation des valeurs dans les templates.
- Les Components, Guards et Pipes doivent appeler les services pour effectuer les traitements.

1.5.2.2. Registre d'entité

Un registre d'entité est en place et permet de gérer les entités métiers de l'application. Pour permettre la réutilisation des instances entités et limité la duplication d'instance en mémoire.

1.5.2.3. Stratégie de detection des changements

Il est préférable d'utiliser la stratégie OnPush sur les composants Angular pour :

- Éviter des fuites CPU.
- Éviter les fuites mémoire au cours de longues sessions d'utilisation.
- Optimiser l'affichage des pages de l'application.

1.5.2.4. Organisation

Une organisation inspirée de la clean architecture est mise en place pour l'application.

```
erp
 ├──── apps
 │    ├──── eshop-client
 │    ├──── crm-client
 │    ├──── ...
 ├──── libs
 │    ├──── web
 │    │    ├──── data
 │    │    │    ├──── crm
 │    │    │    ├──── eshop
 │    │    ├──── feature
 │    │    │    ├──── crm
 │    │    │    ├──── eshop
 │    │    ├──── ui
 │    │    │    ├──── input
 │    │    │    ├──── ...
```

```
|   |   ├──── util-core
|   |   |   ├──── core
|   |   |   ├──── testing
|   |   |   ├──── ...
|   |   ├──── ...
```

Le dossier data correspond aux librairies de services de communication avec le back et de dto associées spécifiques à une fonctionnalité.

Le dossier feature correspond aux librairies de composants spécifiques à une fonctionnalité.

Le dossier util correspond aux librairies de services et de ressources réutilisables.

Le dossier ui correspond aux librairies de composants graphiques réutilisables.

Organisation des fichiers par package

Les fichiers nécessaires au layout principal de l'application seront placées dans un dossier *core*.

Les fichiers des modules métier seront regroupés par modules métier dans un dossier *views*.

Les fichiers réutilisables seront placés dans un dossier *shared*. Les fichiers des entités y seront placés dans un sous-dossier *domains*. Les fichiers des resources y seront placés dans un sous-dossier *resources*. Les fichiers des mappers y seront placés dans un sous-dossier *mappings*.

Les composants seront regroupés par fonctionnalité pour ensuite l'être par type d'objet (component, service, pipe, ...).

1.5.2.5. Formulaires

Les formulaires pendant le lot1 ont été implémentés avec la méthode Template Driven Forms, qui permet une implémentation plus rapide, il était validés avec la librairie Vest (https://github.com/ealush/vest). Cette librairie permet de définir des règles de validation en lien direct avec le domaine métier. Ces règles de validation sont rédigées dans une syntaxe proche des tests unitaires.

Une nouvelle approche est maintenant utilisée pour les formulaires, avec la méthode Reactive Forms. Cette méthode permet une implémentation plus propre et plus maintenable, mais plus longue à mettre en place. Les formulaires seront modifiés au fur et à mesure de qu'une modification doit être apportée.

1.5.3. La partie Back-end détaillée

Le format d'échange JSON entre le front-end et le back-end a été préféré au XML, car il allège les flux réseaux, il est plus compréhensible, et il est plus facilement manipulable par la bibliothèque Angular du front-end.

Le choix du protocole de communication HTTP est un standard des applications web. Cela permet d'assurer une communication avec n'importe quel service/application distant répondant également à ce standard (sous-module VISA, GED, etc.).

Le formalise HATEOS a été mis en hypothèse pour la communication entre le front-end et le back-

end, pour avoir une maitrise des règles métier lié à workflow uniquement côté back-end. Cependant, dans une optique MVP ce choix n'as pas été retenue pour le moment.

1.5.3.1. Gestion de la marque blanche

Une table *organization* est présente dans la base de données, elle permet de gérer les informations de l'entreprise (nom, logo, etc.).

Chaque entité de premier niveau est liée à une entreprise, ce qui permet à cloisonnement pour organization.

1.5.3.2. Organisation des modules métiers

Des modules techniques sont externalisés dans des packages dédiés, pour permettre une meilleure réutilisation et une meilleure maintenabilité, comme la ged et les mails, mais aussi les interfaces de communication avec les providers externes.

Chaque module métier est isolé dans un package java dédié, pour permettre une approche domaniale. Des tests unitaires avec la librairie ArchUnit sont mis en place pour assurer un respect des règles d'architecture établie.

```
erp
 ├──── apps
 │    ├──── eco-micro.auth
 │    ├──── eco-micro.crm
 ├──── libs
 │    ├──── back
 │    │    ├──── eco-micro.mail
 │    │    ├──── eco-micro.trackdechets-interfaces
 │    │    ├──── ...
```

1.5.3.3. Structure de la base de données PostgreSQL

La base de données est composé de deux schémas :

- Le schéma *ecomicro_auth* qui contient les tables de gestion des utilisateurs et des rôles, uniquement accessible par le serveur d'authentification.
- Le schéma *ecomicro_data* qui contient les tables de données.

Lors du démarrage du serveur d'authentification, Liquibase s'exécute sur le schéma ecomicro_auth de l'instance de la DataSource définie en dur dans la configuration.

Il en va de même pour le serveur d'api, Liquibase s'exécute sur le schéma ecomicro_data de l'instance de la DataSource définie en dur dans la configuration.

D'un point de vue structure pour l'écriture des changesets, nous avons un point d'entrée : le fichier db.changelog/master.xml

Les tables de données modifiables par l'utilisateur sont préfixées par *data-*. Les tables de données

non modifiables par l'utilisateur sont préfixées par *ref-*.

1.5.3.4. Accès à la base de données

Deux méthodes sont utilisées pour récupérer les données de la base de données :

- Via Spring Data JPA, qui permet de générer les requêtes SQL en fonction des méthodes définies dans les interfaces des repositories.
- Via Criteria API, qui permet de générer des requêtes SQL en fonction des critères définis dans les classes de critères.

La première méthode est utilisée pour les requêtes simples, la seconde pour les requêtes complexes.

1.5.3.5. Gestion de Open Session In View

L'*Open Session In View* est désactivée pour les requêtes HTTP, car elle peut être la cause de problème de performance.

QueryDSL est utilisé pour la génération des requêtes SQL, car il permet de générer des requêtes SQL en fonction des critères définis dans les classes de critères.

2. Volet développement

Les autres volets du dossier sont accessibles d'ici.

Cette section décrit le code à produire et comment l'écrire.

2.1. Documentation de référence

Table 4. Références documentaires

N°	Version	Titre/URL du document	Détail
1	3.2	https://docs.google.com/document/d/1pMPZekT_XMff5s5Jra1vCpLX1I4NqYWR	La proposition commerciale
2	0.3	https://docs.google.com/document/d/1WuxDCi0bDFyBTOuSaX1i-qgJR6kGl2_lkQ2u3sNbmkU/edit	Les spécifications générales

2.2. Contraintes

Aucune contraintes n'a été recensée.

2.3. Exigences non fonctionnelles

Aucune exigences n'a été recensée.

2.4. Architecture cible

2.4.1. Pile logicielle

Il s'agit d'un front-end *Angular*, avec un back-end Java *Spring Boot*.

La version *Angular* choisie est la dernière release majeure sortie à ce jour, avec comme bibliothèque *@angular/material* pour ces composants graphiques personnalisables et respectant les concepts *Angular*,

RxJs est utilisé pour sa grande versatilité sur les utilitaires liés aux Observables et à la programmation réactive.

Le framework *TailwindCSS* est mise en place pour sa normalisation des classes, il est complété par la technologie Sass pour l'écriture de style plus spécifique au besoin. Parmi les deux syntaxes qu'elle propose, la syntaxe scss est choisie pour la facilité de lecture de son arborescence grâce à ses accolades.

La librairie *ngx-translate* est utilisé pour centraliser les libellés pour permettre de simplifier leurs modifications.

Concernant le back-end, la version *Spring Boot* choisie est la dernière release majeure sortie à ce jour.

La bibliothèque *Spring Cloud OpenFeign* est utilisée pour les appels d'API externes.

Lombok est utilisé pour simplifier l'écriture des classes Java et *Mapstruct* pour simplifier la transformation des objets.

Table 5. Pile logicielle

Librairie	Rôle	Version
Angular	Framework TS pour la conception du front	16.0
Scss	Version de Sass avec accolades	1.49
@angular/material	Framework TS de composants graphiques (modal, tableaux, etc.)	16.0
tailwindcss	Framework CSS	3.2
Rxjs	Framework TS pour la gestion du workflow des Observables	7.8
Karma	Framework de test unitaire	6.4
date-fns	Framework JS de manipulation de dates	2.28
ngx-translate	Framework JS d'internationalisation	14.0
vest	Framework de validation orienté model	14.0
Spring Boot	Framework Java pour la conception du back-end	3.0
Spring Boot Data JPA	Extension Spring Boot pour la gestion d'accès aux données	3.0

Librairie	Rôle	Version
MapStruct	Framework Java pour le mapping de données	1.5
Liquibase	Framework Java pour la maintenance de base de données	4.5
OpenFeign	Framework Java pour la gestion des appels d'API externes	2022.0
Lombok	Framework Java pour l'automatisation de la génération de code	1.18
Spring Authorization Server	Serveur d'authentification Oauth2	1.0
ArchUnit	Test d'architecture du code Java	1.0
Husky	Utilitaire de gestion des hook git	8.0

Archunit, Danger.JS et Husky sont des outils de développement qui ne sont pas utilisés en production. Ils sont utilisés pour la qualité du code et la gestion des MR.

2.4.2. Performances

Quelques règles à suivre lors du développement pour limiter les pertes de performance de l'application.

Concernant le front-end (voir ce document pour plus de détails) :

- Utiliser un *profiler* (comme celui de Chrome) pour détecter les fuite CPU et mémoire.
- Privilégier les appels asynchrones et l'utilisation des Observables avec RxJs.
- Limiter les subscribes dans les composants, privilégier le Pipe async.
- Limiter les conditions, le data binding et l'interpolation à l'utilisation d'une unique variable. Si un traitement ou l'utilisation de plusieurs variables est nécessaire, il est préférable de l'effectuer lors de la récupération des variables concernées, pour stocker la valeur calculée dans une variable publique qui est ensuite injectée dans le template.
- Désabonnement systématique des observables.
- Utilisation de trackBy dans les ngFor si l'item courant dans la boucle est un *object*.
- Éviter les variables globales (implicites et explicites) qui peuvent générer des fuites mémoire et polluer, voire écraser une propriété existante de l'objet global window.
- Éviter les méthodes hors scope.

Concernant le back-end :

- S'assurer que la pagination serveur va bien jusqu'à la base de données (LIMIT, OFFSET).
- Attention aux fonctions SQL qui cassent les index (comme UPPER()). Privilégier les traitements de chaînes de caractères dans le code Java si possible.
- Activer les logs des requêtes SQL (exemple Hibernate : org.hibernate.SQL=DEBUG,-Dhibernate.generate_statistics=true) et vérifier les requêtes SQL et leur nombre (pour détecter

en particulier le problème du SELECT N+1, très courant).

- Vérifier avec un *profiler* la consommation mémoire pour détecter les fuites ou les surconsommations.

Front-end et back-end :

- Traquer le bavardage réseau : grouper les requêtes quand c'est possible.

2.4.3. Spécificités d'usine logicielle

Le gestionnaire de paquets Yarn a été choisi pour la gestion des dépendances front-end car il est plus optimisé et plus stable que npm.

La plate-forme d'intégration continue (*CI*) de GitLab effectue plusieurs étapes à chaque nouveau push sur une branche pour permettre un *feedback* rapide :

- Une étape de *lint*, qui est effectuée par ESLint pour le front-end et par le plugin Apache Maven Checkstyle pour le back-end.
- Une étape de tests unitaires et d'intégration, qui est effectuée par Karma pour le front-end et JUnit pour le back-end.
- Une étape de compilation.

Lors de la création d'un pipeline pour la branche *develop*, une analyse SonarQube est lancée en parallèle du pipeline principal.

Concernant l'analyse SonarQube, plusieurs plugins sont nécessaires :

- Pour le back-end :
  - sonar-maven-plugin (org.sonarsource.scanner.maven:sonar-maven-plugin:3.7.0.1746) : le scanner Sonar pour Maven,
  - jacoco-maven-plugin (org.jacoco:jacoco-maven-plugin:0.8.6) : bibliothèque d'analyse de la couverture de code Java.
- Pour le front-end :
  - sonarqube-scanner@2.8.1 : le scanner Sonar pour Node.js,
  - karma-sonarqube-unit-reporter@0.0.23 : le rapporteur Karma pour générer le rapport lcov pour indiquer la couverture de code à Sonar.

2.4.4. Normes de développement et qualimétrie

Deux méthodologies sont appliquées :

- le TDD : voir l'annexe C pour plus de détails,
- le Clean Code : voir l'annexe B pour plus de détails.

Le niveau de qualité exigé correspond au Quality Gate SonarQube recommandé :

- 80% de couverture de code minimum, hors accesseurs (*getters*),

- 3 % maximum de lignes dupliquées,
- Niveau A en *Maintenabily*, *Relability* et *Security*.

Le code, ses commentaires et les messages de commits sont écrits en anglais, à l'exception des termes métiers difficilement traduisibles. La documentation, le titre des merges requests, et leurs commentaires sont en francais.

Les commentaires sont obligatoires sur les fonctions. Ils sont optionnels sur les fonctions de test et les propriétés.

Les mappeurs doivent rester des interfaces et peuvent utiliser des @QualifiedBy au besoin. Ils ne doivent pas avoir d'effet de bord sur la base de données (pas d'insert/update/delete).

L'indentation du code est faite avec 4 espaces, ce qui permet une meilleure uniformisation entre les différents supports de visualisation (GitLab, IntelliJ IDEA, terminal, etc.).

2.4.5. Patterns notables

2.4.5.1. [Front] Pattern Smart/Dumb

Pour séparer un maximum l'aspect logique métier, de l'aspect logique d'affichage de l'information, les composants sont séparés en deux types principaux :

- Les composants "Smart" (composant d'entrée des modules, etc.) qui récupèrent les données venant des services et les redistribuent aux composants "Dumb" ou "Smart" qui les composent.
- Les composants "Dumb" qui se chargent d'afficher les données : composants de listes, composants de tableaux, etc.

2.4.6. Spécificités des tests

Dans une optique MVP, aucun tests E2E n'est prévu pour le moment.

Voici la liste des types de test attendus :

Table 6. Types de tests

Type de test	Temps à investir	Manuel ou automatisé ?	Type de module ciblé	Taux de Couverture visée	Détail
TU	Très élevé	Automatisé	Back-end et front-end	env. 80%	Réalisés lors du développement en TDD, écrits sous la forme Given, When, Then.

Type de test	Temps à investir	Manuel ou automatisé ?	Type de module ciblé	Taux de Couverture visée	Détail
TI	Faible	Automatisé	Liens UI/API	env. 100% du code appelant côté UI et des contrôleurs Spring côté API	Teste la non régression des échanges lors de l'appel des opérations des API REST (principe CDC=Consumer-Driven Contract) via les outils Pact et pact-react-consumer.
Tests système	Faible	Manuels	UI	10%	Tests menés par l'équipe de développement couvrant des scénarios fonctionnels complets. Le but est ici de tester le fonctionnement de l'ensemble des modules (ce qui n'est pas automatisable) et de détecter un maximum de bugs avant les tests d'UAT.
Tests UAT (acceptance)	Moyen	Manuels	UI	de 30% à 80% selon le nombre de scénarios prévus	Tests menés en recette par des utilisateurs finaux sur environnement non bouchonné.

Les tests d'intégration et les tests unitaires, sont positionnés dans deux modules séparés, pour diminuer le temps d'exécution des tests.

Les tests unitaires sont très rapides à exécuter, car ils ne chargent pas le context d'exécution spring. Cela permet aux développeurs d'exécuter rapidement tous les tests unitaires avant un *commit* de modifications.

Les tests d'intégration sont plus lents et moins nombreux que les tests unitaires. Ils impliquent toutes les couches de l'application : de la couche REST la plus haute à la base de données.

La nomenclature des tests java est la suivante : `{method}_{cas de test}\_{résultat attendu}` (e.g : getAll_whenNotXNameSchemaHeaderProvided_returnError)

2.4.7. Gestion de la robustesse

2.4.7.1. Gestion des sessions

L'application est SessionLess, c'est-à-dire qu'elle ne conserve pas de session d'utilisateur. Les différents paramètres nécessaires à l'identification de l'utilisateur sont fournis en entêtes de chaque requête HTTP.

2.4.7.2. Gestion des erreurs

Les erreurs techniques (imprévues) survenant lors de l'appel à un service REST comme un *timeout* sont catchées au plus haut niveau de l'application (via un ErrorHandler). Toutes les informations sont journalisées avec la stack-trace complète, mais l'appelant ne récupère qu'un code d'erreur générique sans détails techniques (pour raison de sécurité).

Les retours de valeur à null sont évités. L'émission d'une exception est préférée.

Quand une exception est catchée, elle ne doit pas simplement être automatiquement journalisée et renvoyée, il est préférable de tenter de récupérer l'erreur ou de faire un *retry* si possible.

Côté front-end, les erreurs des observables sont toujours attrapées (*catch*) . Si un utilisateur effectue une action qui produit une erreur, il obtient un retour l'informant que sa demande n'a pas pu être correctement traitée.

Côté back-end, les erreurs sont gérées à l'aide de problèmes, décrits dans la RFC 7807 *Problem Details for HTTP APIs*, en utilisant l'implémentation mise en place par Spring Boot. Un problème permet de modéliser une erreur, dans un format standard, avec un message d'erreur clair.

La bibliothèque utilisée permet de configurer de façon transverse la journalisation des erreurs :

- les erreurs ayant un code HTTP entre 400 et 499 (inclus) sont journalisées par défaut avec le niveau WARN,
- les erreurs ayant un code HTTP supérieur ou égal à 500 sont journalisées par défaut avec le niveau ERROR.

Le niveau de journalisation d'une instance spécifique d'un problème peut-être modifié en affectant la variable `logLevel` du problème (à WARN pour forcer un niveau WARN, à ERROR pour forcer un niveau ERROR). Le code retour HTTP n'est pas modifié par les modifications de niveau d'erreur.

Le message d'erreur retourné lors des appels aux APIs REST est la représentation au format JSON du problème survenu, et contient un champ *errorCode* dont la valeur est une chaîne de caractères. Ce code d'erreur est utilisé par le front-end pour afficher à l'utilisateur un message compréhensible.

Le code d'erreur contient 3 ou 4 parties séparées par un point (`.`) :

- le module ou domaine,
- le sous-domaine ou la fonctionnalité concernée,
- la sous-fonctionnalité concernée,
- l'erreur survenue.

Dans le cas où l'erreur est rattachée directement à une fonctionnalité, la 3^{me} partie n'est pas présente, et le code d'erreur ne contient que deux points.

2.4.8. Gestion de la configuration

Plusieurs configurations initiales seront présentes lors de la compilation des conteneurs, et elles

pourront être activées par sélection de profils actifs. La configuration est injectée au lancement de l'application en définissant des variables d'environnements dans un fichier docker-compose.yml.

2.4.9. Politique de gestion des branches

Table 7. Types de branches

Branche	Nombre	Branche d'origine	Durée de vie	Fonction
develop	1		A vie	Code prêt à être envoyé pour recette
release/###	n	develop	Jusqu'a livraison	Code prêt à être envoyé pour recette et en production
feature/###	n	develop	Temps du developpment	Développement d'une fonctionnalité
bugfix/###	n	develop	Temps du developpment	Correction d'un bug interne
hotfix/###	n	develop	Temps du developpment	Correction d'un bug en production

La procédure de merge request (MR) est explicitée ci-dessous :

- Création de la MR à partir du modèle de *MR_Template*.
- Suivi du modèle de MR : cocher les cases au fur et à mesure.
- Revue de la MR par un autre développeur. Le relecteur ouvre un commentaire pour chaque remarque concernant le code relu. Lorsqu'un relecteur est en accord avec les changements, il approuve la MR en cliquant sur le bouton *Approve*.
- L'auteur d'un commentaire de MR doit le marquer comme résolu une fois en accord avec les modifications apportées par le développeur ayant proposé la MR.
- L'auteur de la MR est en charge du merge quand il a reçu au moins une approbation (*approval*).

2.4.10. Gestion du code source et versionnage

D'une façon générale, toute ressource (source, outil, script de CI/CD, template, DDL de base de données, DAT, etc.) doit être gérée à l'aide d'un logiciel de gestion de version. Le logiciel de gestion de version utilisé pour le projet est git. La forge utilisée pour le projet est GitLab.

Le front-end et le back-end sont versionnés et livrés sous la forme d'une image docker sur le registry de GitLab.

Les modules seront versionnés suivant la numérotation x.y.z (<majeur>.<mineur>.<correctif>).

2.4.11. Gestion de la concurrence

Tous les contrôleurs ont le *scope* singleton et ne doivent donc en aucun cas stocker d'état dans leurs attributs pour éviter des *race conditions*.

2.4.12. Encodage

L'application utilisera l'encodage "UTF-8".

2.4.13. Fuseaux horaires

L'application sera fixée sur le fuseau horaires "Europe/Paris".

2.4.14. Gestion des logs

Plusieurs règles d'ordre général sont définies :

- Ne pas laisser de logs de développement dans le code (exemple : console.out("entrée dans méthode x"))
- Les logs contiennent des chaînes de caractère discriminantes (exemple : code erreur) pour faciliter le filtrage dans l'outil de recherche de logs.
- Les logs contiennent des identifiants d'entités permettant de retrouver l'objet concerné.

Les logs de l'application sont stockés dans le fichier *ecomicro-app.YYYY-MM-DD.log* situé dans le répertoire de lancement de l'application. Il s'agit d'un fichier texte généré par le logger de l'application. Une rotation de ce fichier est automatique faite à chaque première utilisation du jour.

2.4.15. Tri et Pagination

Les requêtes de listage d'éléments sont systématiquement triées selon un ordre ascendant (le défaut) ou descendant. De plus, il sera possible de choisir le champ sur lequel se fait le tri via le query param `sort`.

Afin de limiter le nombre d'objets à destination du front-end, l'API retourne un nombre limité d'éléments (ce nombre sera paramétrable suivant la taille des éléments individuels).

Il s'agit des query params `page` et `size` contenant le numéro de la page à récupérer et le nombre d'éléments de la page.

Chaque API concernée propose des valeurs par défaut pour ces trois paramètres.

2.4.16. Provisioning et mises à jour des DDL

Nous utilisons Liquibase embarqué dans le jar de l'application pour créer et mettre à jour les DDL de la base.

Aucun scripts SQL ne doit être lancés. Les requêtes nécessaires sont effectuées directement par l'application lors de son démarrage.

3. Volet infrastructure

Les autres volets du dossier sont accessibles d'ici.

Cette section décrit le déploiement des modules applicatifs dans leur environnement d'exécution cible et l'ensemble des dispositifs assurant leur bon fonctionnement.

3.1. Documentation de référence

Table 8. Références documentaires

N°	Version	Titre/URL du document	Détail
1	3.2	https://docs.google.com/document/d/1pMPZekT_XMff5s5Jra1vCpLX1I4NqYWR	La proposition commerciale
2	0.3	https://docs.google.com/document/d/1WuxDCi0bDFyBTOuSaX1i-qgJR6kGl2_lkQ2u3sNbmkU/edit	Les spécifications générales

3.2. Contraintes

Une contrainte a été identifiée :

- Tout le trafic réseau d'un port ne peut être redirigé que vers une seule machine.

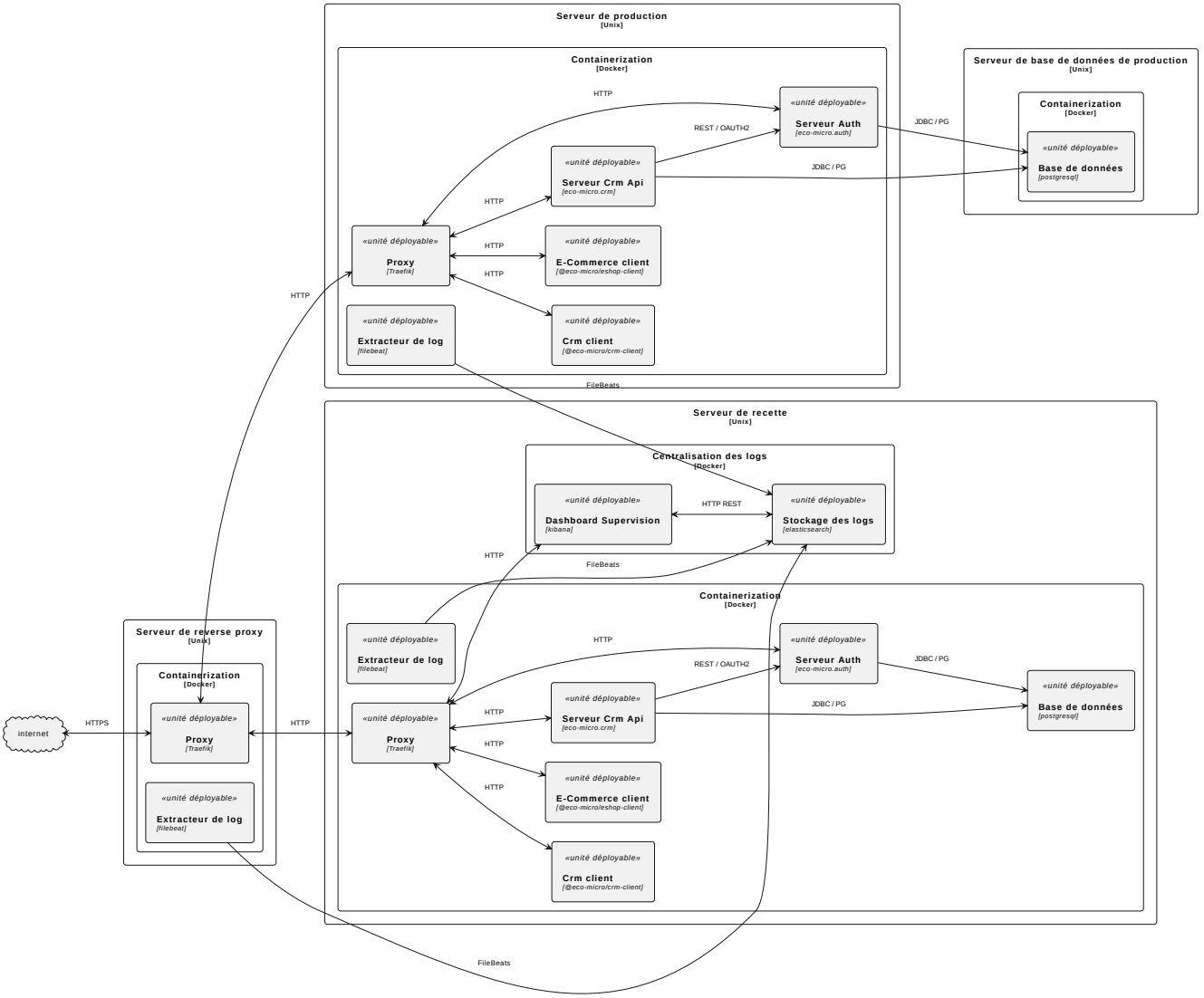
3.3. Exigences

Les exigences influant sur l'infrastructure sont les suivantes :

- L'application doit être conteneurisée.
- Les environnements de recette et de production doivent être séparés.
- La base de données de production doit être externalisé.

3.4. Architecture cible

La figure ci-dessous représente l'infrastructure mise en place.



Une pile ELK est utilisée pour centraliser et analyser les logs.

Les different services sont déployés dans des conteneurs docker séparés pour permettre une mise à jour indépendante de chaque service.

Le *Proxy* du *Serveur de reverse proxy* est l'unique point d'entrée de la solution.

3.4.1. Versions des composants

Table 9. Composants d'infrastructure

Composant	Rôle	Version	Environnement technique
Elasticsearch	*Stack Elasticsearch / Logstash / Kibana*	*elasticsearch:8.9.0*	*Docker*
Kibana	*Stack Elasticsearch / Logstash / Kibana*	*kibana:8.9.0*	*Docker*
FileBeat	*Stack Elasticsearch / Logstash / Kibana*	*filebeat-oss:8.9.0*	*Docker*

Composant	Rôle	Version	Environnement technique
Java	*Serveur d'application JEE*	*eclipse-temurin:17-jre-jammy*	*Docker*
Traefik	*Proxy*	*traefik:v2.5*	*Docker*
PostgreSQL	*SGBD-relationnelle*	*postgres:13.4-alpine*	*Docker*

4. Volet dimensionnement

Les autres volets du dossier sont accessibles d'ici.

Cette section décrit le dimensionnement du projet. Le but est de déterminer la taille de l'infrastructure nécessaire au projet.

4.1. Documentation de référence

Table 10. Références documentaires

N°	Version	Titre/URL du document	Détail
1	3.2	https://docs.google.com/document/d/1pMPZekT_XMff5s5Jra1vCpLX1I4NqYWR	La proposition commerciale
2	0.3	https://docs.google.com/document/d/1WuxDCi0bDFyBTOuSaX1i-qgJR6kGl2_lkQ2u3sNbmkU/edit	Les spécifications générales

4.2. Contraintes

Aucune contraintes n'a été recensée.

4.3. Exigences

Les exigences influant sur le dimensionnement sont les suivantes :

- Les environnements de recette et de production doivent être séparés.
- La base de données de production doit être externalisé.

4.4. Dimensionnement

4.4.1. Estimation des besoins en ressources par composant technique

Table 11. Estimation des besoins en ressources par composant technique

Profil	Mémoire
Production - PostgreSQL	32Go
Production - JVM Authentification	16Go
Production - JVM Api	32Go
Recette - PostgreSQL	8Go
Recette - JVM Authentification	8G
Recette - JVM Api	16Go
Monitoring - JVM Elasticsearch	32Go

4.4.2. Dimensionnement des machines

Table 12. Dimensionnement des machines

Type de machine	Nb de machines	Nb (V)CPU	Mémoire (Gio)	Disque interne (Go)
Serveur de base de données	1	10	32	1000
Serveur applicatif	1	6	64	100
Serveur de monitoring et recette	1	8	32	1000

5. Volet sécurité

Les autres volets du dossier sont accessibles d'ici.

Cette section décrit l'ensemble des dispositifs mis en œuvre pour empêcher l'utilisation non-autorisée, le mauvais usage, la modification illégitime ou le détournement des modules applicatifs.

5.1. Documentation de référence

Table 13. Références documentaires

N°	Version	Titre/URL du document	Détail
1	3.2	https://docs.google.com/document/d/1pMPZekT_XMff5s5Jra1vCpLX1I4NqYWR	La proposition commerciale
2	0.3	https://docs.google.com/document/d/1WuxDCi0bDFyBTOuSaX1i-qgJR6kGl2_lkQ2u3sNbmkU/edit	Les spécifications générales

5.2. Contraintes

Les contraintes suivantes ont été identifiées :

- Une même adresse mail ne peut être utilisée pour créer plusieurs comptes.

- Un compte a accès à une seule application (crm/eshop).

5.3. Exigences

Les exigences suivantes ont été identifiées :

- Le protocole Oauth2 est à utiliser pour l'authentification des utilisateurs.
- Les mots de passe doivent être stockés sous forme de digest bcrypt.
- Les mots de passe doivent être suffisamment complexes.

5.4. Mesures de sécurité

Un Spring Authorization Server est en place pour externaliser l'authentification. Il soit le process décrit dans la documentation de référence de l'IETF pour les SPA section [6.3].

Une attention particulière est à porter, car le client utilisé est un client public. Le *grant type client-credentials* ne lui est pas accordé.

La norme JWT est utilisée pour la réalisation du token.

5.4.1. Intégrité

Les entités sont référencées par des ID et par leur version.

5.4.2. Confidentialité

L'accès à la liste des clients et à la liste des affaires nécessite une authentification réussie par login/mot de passe d'un utilisateur ayant le role INTERNE.

5.4.3. Identification

Les utilisateurs sont identifiés par leurs addresses mail.

5.4.4. Authentification

L'authentification des internautes inscrits se fait par login/mot de passe (respectant la politique de mot de passe mis en place).

Les mots de passe ne sont pas conservés en clair, mais stockés sous la forme de digest bcrypt.

5.4.5. Fédération d'identité

N/A

5.4.6. SSO, SLO

N/A

5.4.7. Non-répudiation

N/A

5.4.8. Anonymat et vie privée

Un consentement explicite des utilisateurs dans la conservation de leurs données personnelles est proposé à la création du compte depuis le site e-commerces.

La base de données pour test est anonymisée.

5.4.9. Habilitations

La gestion des habilitations est effectuée par le serveur d'authentification.

Il s'agit d'un système d'autorisation par ressource, chaque utilisateur se voit attribuer un ensemble de rôles, et chaque rôle se voit attribuer un ensemble de permissions sur des resources.

[Role]

L'ensemble des utilisateurs et des rolesEntity sont stockées dans la base de données PostgreSQL, dans un schema séparé des données applicatives. Seul le serveur d'authentification a accès à ce schéma.

Les permissions seront composées de deux parties, le nom de la ressource et le niveau de droit associée.

Trois niveaux de droit seront disponibles pour chaque resource :

- READ : lecture des données
- WRITE : écriture des données
- EXECUTE : suppression des données

Les niveaux de droit sont cumulatifs, un utilisateur ayant le droit WRITE sur une ressource peut avoir le droit READ sur cette même ressource.

Plusieurs rôles sont mis en place par défaut :

- ADMINISTRATEUR : donne accès à l'ensemble des permissions de paramétrage de l'application
- RESPONSABLE
- COMMERCIAL
- CONSULTANT
- USER : role réservé au utilisateur du E-commerce.

5.4.10. Tracabilité, auditabilité

- Chaque requête REST est stockée dans une base ElasticSearch pour consultation en Kibana, grâce à la pile ELK.

- Un *header* RequestId est rajouté en en-tête des requêtes, pour permettre une corrélation des requêtes effectuées.

5.5. Auto-contrôles

5.5.1. Auto-contrôle des vulnérabilités

Table 14. Checklist d'auto-contrôle de prise en compte des vulnérabilités courantes

Vulnérabilité	Pris en compte ?	Mesures techniques entreprises
Accès à des ports privés	X	Configuration du pare-feu iptables sur la machine exposée à Internet. Seuls le port 443 est ouvert.
Attaque de mot de passe par force brute		
Injection SQL	X	Utilisation de PreparedStatement uniquement, audit des requêtes SQL.
Injection OS	X	Vérification qu'il n'y a aucun appel de commandes systèmes dans le code (type Runtime.exec())
Violation de gestion d'authentification et de session	X	Traité avec un dispositif anti-CSRF.
XSS	X	• *Spécification systématique de l'encoding dans le header de réponse Content-Type (ex : text/html;charset=UTF-8) pour parer les attaques basées sur des caractères spéciaux contournant l'anti-XSS* ◦ Utilisation d'une librairie d'échappement. Pour les modules Java, la méthode StringEscapeUtils.escapeHtml4() de org.apache.commons.text est utilisée. ◦ Spécification systématique de l'encodage dans l'entête de réponse Content-Type (ex : text/html;charset=UTF-8) pour parer les attaques basées sur des caractères spéciaux contournant l'anti-XSS.
Référence directe à un objet	X	Vérification à chaque requête que les arguments passés correspondent bien à la personne identifiée. Par exemple, toute requête contient son ID et on vérifie par une requête que l'objet qu'elle tente de consulter lui appartient bien avant de poursuivre la requête.
Planification des mises à jour de sécurité		

Exposition de données sensibles	X	• Tous les algorithmes de sécurité sont à jour : au minimum SHA-256, AES 256. • L'application ne fonctionne qu'en HTTPS.
CSRF	X	Utilisation d'un dispositif anti-CSRF.
Log injection	X	Échappement des logs avant de les transmettre à log4j.
Attaques HTTPS + compression CRIME/BREACH	X	Dispositif anti-CSRF.
Téléversement de fichiers malicieux	X	Limitation sur le format des fichiers téléversés.

5.5.2. Auto-contrôle RGPD

Table 15. Checklist d'auto-contrôle de respect du RGPD

Exigence RGPD	**Prise en compte ?**	**Mesures techniques entreprises**
Registre du traitement de données personnelles		
Pas de données personnelles inutiles		
Droits des personnes (information, accès, rectification, opposition, effacement, portabilité et limitation du traitement.)		
Sécurisation des données		

Glossaire des termes utilisés

Merci de respecter l'ordre alphabétique lors des éditions du document

Terme	Signification	Commentaire
Bibliothèque	Un ensemble de fonctions utilitaires, regroupées et mises à disposition afin de pouvoir être utilisées sans avoir à les réécrire.	

Annexe A: Clean Code

A.1. Introduction

Code is clean if it can be understood easily – by everyone on the team. Clean code can be read and enhanced by a developer other than its original author. With understandability comes readability, changeability, extensibility and maintainability.

A.2. General rules

1. Follow standard conventions.
2. Keep it simple stupid. Simpler is always better. Reduce complexity as much as possible.
3. Boy scout rule. Leave the campground cleaner than you found it.
4. Always find root cause. Always look for the root cause of a problem.

A.3. Design rules

1. Keep configurable data at high levels.
2. Prefer polymorphism to if/else or switch/case.
3. Separate multi-threading code.
4. Prevent over-configurability.
5. Use dependency injection.
6. Follow Law of Demeter. A class should know only its direct dependencies.

A.4. Understandability tips

1. Be consistent. If you do something a certain way, do all similar things in the same way.
2. Use explanatory variables.
3. Encapsulate boundary conditions. Boundary conditions are hard to keep track of. Put the processing for them in one place.
4. Prefer dedicated value objects to primitive type.
5. Avoid logical dependency. Don’t write methods which works correctly depending on something else in the same class.
6. Avoid negative conditionals.

A.5. Names rules

1. Choose descriptive and unambiguous names.
2. Make meaningful distinction.
3. Use pronounceable names.
4. Use searchable names.
5. Replace magic numbers with named constants.

6. Avoid encodings. Don’t append prefixes or type information.

A.6. Functions rules

1. Small.
2. Do one thing.
3. Use descriptive names.
4. Prefer fewer arguments.
5. Have no side effects.
6. Don’t use flag arguments. Split method into several independent methods that can be called from the client without the flag.

A.7. Comments rules

1. Always try to explain yourself in code.
2. Don’t be redundant.
3. Don’t add obvious noise.
4. Don’t use closing brace comments.
5. Don’t comment out code. Just remove.
6. Use as explanation of intent.
7. Use as clarification of code.
8. Use as warning of consequences.

A.8. Source code structure

1. Separate concepts vertically.
2. Related code should appear vertically dense.
3. Declare variables close to their usage.
4. Dependent functions should be close.
5. Similar functions should be close.
6. Place functions in the downward direction.
7. Keep lines short.
8. Don’t use horizontal alignment.
9. Use white space to associate related things and disassociate weakly related.
10. Don’t break indentation.

A.9. Objects and data structures

1. Hide internal structure.
2. Prefer data structures.
3. Avoid hybrids structures (half object and half data).
4. Should be small.
5. Do one thing.
6. Small number of instance variables.
7. Base class should know nothing about their derivatives.
8. Better to have many functions than to pass some code into a function to select a behavior.
9. Prefer non-static methods to static methods.

A.10. Tests

1. One assert per test.
2. Readable.
3. Fast.
4. Independent.
5. Repeatable.

A.11. Code smells

1. Rigidity. The software is difficult to change. A small change causes a cascade of subsequent changes.
2. Fragility. The software breaks in many places due to a single change.
3. Immobility. You cannot reuse parts of the code in other projects because of involved risks and high effort.
4. Needless Complexity.
5. Needless Repetition.
6. Opacity. The code is hard to understand.

Annexe B: TDD

B.1. Introduction

This document shows the Test-driven development.

B.2. Test-driven development cycle

- **Add a test**
  - The adding of a new feature begins by writing a test that passes iff the feature's specifications are met. The developer can discover these specifications by asking about use cases and user stories. A key benefit of test-driven development is that it makes the developer focus on requirements before writing code. This is in contrast with the usual practice, where unit tests are only written after code.
- **Run all tests The new test should fail for expected reasons**
  - This shows that new code is actually needed for the desired feature. It validates that the test harness is working correctly. It rules out the possibility that the new test is flawed and will always pass.
- **Write the simplest code that passes the new test**
  - Inelegant or hard code is acceptable, as long as it passes the test. The code will be honed anyway in Step 5. No code should be added beyond the tested functionality.
- **All tests should now pass**
  - If any fail, the new code must be revised until they pass. This ensures the new code meets the test requirements and does not break existing features.
- **Refactor as needed, using tests after each refactor to ensure that functionality is preserved**
  - Code is refactored for readability and maintainability. In particular, hard-coded test data should be removed. Running the test suite after each refactor helps ensure that no existing functionality is broken.
    - Examples of refactoring:
      - moving code to where it most logically belongs
      - removing duplicate code
      - making names self-documenting
      - splitting methods into smaller pieces
      - re-arranging inheritance hierarchies
- **Repeat**
  - The cycle above is repeated for each new piece of functionality. Tests should be small and incremental, and commits made often. That way, if new code fails some tests, the programmer can simply undo or revert rather than debug excessively. When using external libraries, it is important not to write tests that are so small as to effectively test merely the library itself, unless there is some reason to believe that the library is buggy or not feature-rich enough to serve all the needs of the software under development.

B.3. Best practices

B.3.1. Test structure

Effective layout of a test case ensures all required actions are completed, improves the readability of the test case, and smooths the flow of execution. Consistent structure helps in building a self-documenting test case. A commonly applied structure for test cases has (1) setup, (2) execution, (3) validation, and (4) cleanup.

- Setup: Put the Unit Under Test (UUT) or the overall test system in the state needed to run the test.
- Execution: Trigger/drive the UUT to perform the target behavior and capture all output, such as return values and output parameters. This step is usually very simple.
- Validation: Ensure the results of the test are correct. These results may include explicit outputs captured during execution or state changes in the UUT.
- Cleanup: Restore the UUT or the overall test system to the pre-test state. This restoration permits another test to execute immediately after this one. In some cases in order to preserve the information for possible test failure analysis the cleanup should be starting the test just before the test's setup run.

B.3.2. Individual best practices

Some best practices that an individual could follow would be to separate common set-up and tear-down logic into test support services utilized by the appropriate test cases, to keep each test oracle focused on only the results necessary to validate its test, and to design time-related tests to allow tolerance for execution in non-real time operating systems. The common practice of allowing a 5-10 percent margin for late execution reduces the potential number of false negatives in test execution. It is also suggested to treat test code with the same respect as production code. Test code must work correctly for both positive and negative cases, last a long time, and be readable and maintainable. Teams can get together with and review tests and test practices to share effective techniques and catch bad habits.

B.3.3. Practices to avoid, or "anti-patterns"

- Having test cases depend on system state manipulated from previously executed test cases (i.e., you should always start a unit test from a known and pre-configured state).
- Dependencies between test cases. A test suite where test cases are dependent upon each other is brittle and complex. Execution order should not be presumed. Basic refactoring of the initial test cases or structure of the UUT causes a spiral of increasingly pervasive impacts in associated tests.
- Interdependent tests. Interdependent tests can cause cascading false negatives. A failure in an early test case breaks a later test case even if no actual fault exists in the UUT, increasing defect analysis and debug efforts.
- Testing precise execution behavior timing or performance.
- Building "all-knowing oracles". An oracle that inspects more than necessary is more expensive and brittle over time. This very common error is dangerous because it causes a subtle but pervasive time sink across the complex project.[9]

- Testing implementation details.
- Slow running tests.

Annexe C: Best Practices Angular

C.1. Introduction

This document show a few best practices with angular.

C.2. Generale Rules

- Per file, the code must not exceed from 400 lines limit
- Per function, the code must not exceed from 75 lines
- If the values of the variables are intact, declare it with const
- Names of properties and methods should be in lower camel case
- We shouldn't name our interfaces with the starting capital I letter as we do in some programming languages.
- Breaking down Components : This might be an extension of the single responsibility principle not just to the code files or the methods, but to components as well. The larger the component is, the harder it becomes to debug, maintain and test.
- Using Interfaces
- Safe Navigation Operator (?) : To be on the safe side we should always use the safe navigation operator while accessing a property from an object in a component's template. If the object is null and we try to access a property, we are going to get an exception. But if we use the save navigation (?) operator, the template will ignore the null value and will access the property once the object is not null anymore.
- Prevent Memory Leaks in Angular Observable : Observable memory leaks are very common and found in every programming language, library, or framework. Angular is no exception to that. Observables in Angular are very useful as it streamlines your data, but memory leak is one of the very serious issues that might occur if you are not focused. It can create the worst situation in mid of development. Here're some of the tips which follow to avoid leaks.
    - Using async pipe
    - Using take(1)
    - Using takeUntil()
- Using index.ts : index.ts helps us to keep all related things together so that we don't have to be bothered about the source file name. This helps reduce the size of the import statement.
- Change Detection Optimisations :
    - Use NgIf and not CSS - If DOM elements aren't visible, instead of hiding them with CSS, it's a good practice to remove them from the DOM by using *ngIf.
    - Move complex calculations into the ngDoCheck lifecycle hook to make your expressions

faster.

    - Cache complex calculations as long as possible
    - Use the OnPush change detection strategy to tell Angular there have been no changes. This lets you skip the entire change detection step.

- Build Reusable Components : If there is a piece of UI that you need in many places in your application, build a component out of it and use the component. This will save you a lot of trouble if, for some reason, the UI has to be changed. In that case, you do not go around changing the UI code in all the places. Instead, you can change the code in the component and that is it.
- Using trackBy in NgFor : When using ngFor to loop over an array in templates, use it with a trackBy function which will return a unique identifier for each DOM item. When an array changes, Angular re-renders the whole DOM tree. But when you use trackBy, Angular will know which element has changed and will only make DOM changes only for that element.
- Using Smart - Dumb components : This pattern helps to use OnPush change detection strategy to tell Angular there have been no changes in the dumb components. Smart components are used in manipulating data, calling the APIs, focussing more on functionalities, and managing states. While dumb components are all about cosmetics, they focus more on how they look.
- Using strict types instead of "any" : While working on an Angular project, developers, generally end up typing ‘any’ to declare variables. If you are not specifying the variables and constants, they will be assumed by the value and as a result, will be assigned to it. If it happens, you are now in trouble as it will create some unintended issues, anytime.