PREMIER
MINISTRE

Liberté
Égalité
Fraternité

Cadre de cohérence technique

Modèles de Conception et Cadre d'Implémentation

DSAF
DIRECTION DES SERVICES ADMINISTRATIFS ET FINANCIERS DU PREMIER MINISTRE

Table des matières

1. Contexte 4

1.1. Objet 4

2. Cadre d'implémentation 4

2.1. Généralités 4

2.2. Conteneurs 5

3. Exigences 5

3.1. Supervision et observabilité 5

3.1.1. Principe de fonctionnement 5

3.1.2. Exigences 8

3.2. Analyse antivirus 8

3.3. Détection des secrets 8

3.4. Gestion des vulnérabilités 9

3.5. Déploiement en production 9

4. Normes de développement 9

4.1. Généralités 9

4.2. Principe du « Clean Code » 10

4.3. Suivi de version 10

4.4. Stratégie et exigences de tests 11

4.4.1. Typologie de tests 11

4.5. Web 12

4.6. API 12

4.6.1. Principe de fonctionnement 12

4.6.2. Critères d'implémentation 13

4.6.3. Exigences 13

4.7. Javascript 15

4.8. Java 15

4.9. PHP 15

4.10. Python 15

1. Contexte

1.1. Objet

L'objectif de ce document est de définir les patrons de conception et les règles de développement à respecter dans le cadre de l'architecture du système d'information des services du Premier ministre.

2. Cadre d'implémentation

2.1. Généralités

En référence à la circulaire du 19 Septembre 2012 signée par le Premier ministre, l'utilisation de logiciel libre est à privilégier dans le développement des systèmes d'information de l'Etat.

L'ensemble des logiciels libres préconisés se présente sous la forme du socle interministériel de logiciels libres (SILL).

En particulier, concernant les principaux socles applicatifs :

- Java
- PHP

Les composants de référence, à l'exception du lot 2, sont les suivants :

Composant	
Système d'exploitation serveur	Version stable de Debian
Développement d'application Web Java	SpringBoot* LTS OpenJDK LTS ou suivant la version dans la distribution stable de debian
Développement d'application Web PHP	Symfony* LTS PHP en suivant la version dans la distribution stable de debian
Serveur Web	Apache en suivant la version dans la distribution stable de debian
Serveur d'application Java	Tomcat en suivant la version

	dans la distribution stable de debian
Base de données	Postgresql – version incluse dans la distribution stable Debian

*non inclus dans les repository Debian

De manière générale, il convient de suivre la version stable de Debian et d'utiliser les versions des composants embarqués par la distribution Debian dans une version donnée supportée.

De même, il convient de privilégier les déploiements des composants au travers des repositories officiels de Debian, plutôt que des installations à base de paquets externes.

Si toutefois l'application nécessite, pour des raisons de compatibilité un socle différent de Debian, il conviendra de s'astreindre aux mêmes objectifs de compatibilités des composants avec les repositories officiels de la distribution.

Toute fourniture de conteneur se basant sur une base différente de Debian nécessite une validation du pouvoir adjudicateur.

Tout paquet livré en dehors de la distribution doit être maintenu et packagé par le titulaire. Des paquets externes pourront être refusés notamment pour des raisons de sécurité ou d'exploitabilité.

Toute livraison doit être autosuffisante, et ne pas faire appel à des ressources externes, ni contenir de paquets obsolètes.

2.2. Conteneurs

En cas d'utilisation de conteneurs, il convient d'utiliser Podman ou Docker en configuration rootless. Un conteneur applicatif doit être « stateless », pas de stockage en interne du conteneur.

Les Recommandations de sécurité relatives au déploiement de conteneurs Docker du guide de l'ANSSI sont à appliquer.

Dans le cadre d'une livraison de conteneurs, la livraison doit contenir tous les éléments (Dockerfile, fichiers d'env, docker-compose.yml…) afin que la DSAF/DSI puisse être autonome dans la construction et le déploiement.

3. Exigences

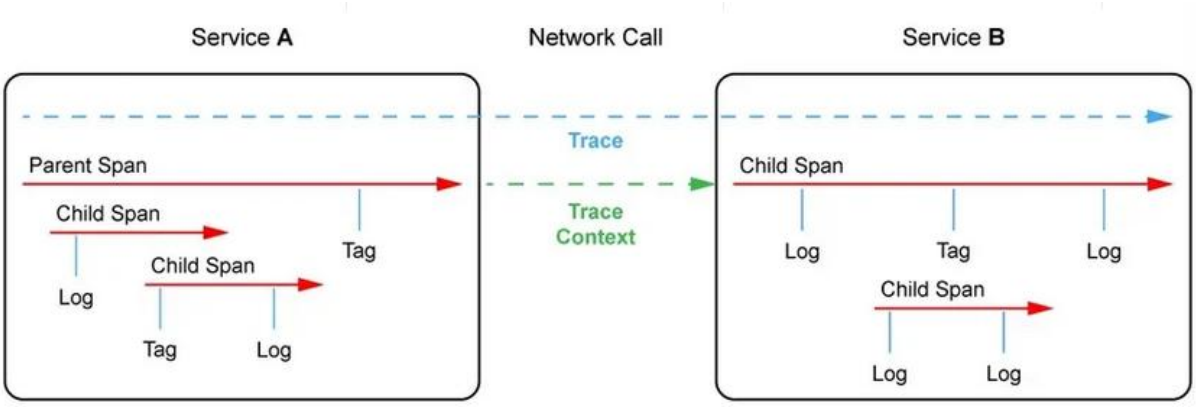
3.1. Supervision et observabilité

3.1.1. Principe de fonctionnement

Afin de comprendre le comportement du système, d'anticiper d'éventuelles erreurs et de prévenir des coupures de service, nous devons être en mesure de suivre le parcours complet d'une requête à travers tous les composants qu'elle traverse. Ainsi, les applications devront émettre des données télémétriques qui seront ensuite collectées et ingérées dans des serveurs de supervision pour alimenter des tableaux de bord.
Les données télémétriques sont principalement :

- **Les journaux (logs)** : ils représentent des messages émis par chaque application au fil du traitement. Un journal apporte du détail fonctionnel, le message d'erreur exact, la cause.
- **Le span (segment)**: il représente une unité de traitement au sein de la trace, par exemple une requête d'API, une requête en base de données, la publication d'un événement asynchrone. Chaque span comporte une durée, un statut, des attributs meta data, et la référence du span parent permettant de reconstruire l'arborescence complète du traitement.
- **La trace** : elle représente le parcours de bout en bout d'une requête. Il s'agit d'une collection de spans.

Le diagnostic d'un incident suit donc ce chaînage : la trace montre le parcours et son chronogramme, le span en anomalie localise le composant et l'étape, les journaux corrélés donnent la cause.



Source : OpenTelemetry.io

Tout nouveau code produit doit être accompagné de journaux permettant une traçabilité fine des actions de bout en bout. Le niveau de journalisation doit être paramétrable et documenté, avoir une granularité par composants si besoin, et la destination des journaux peut être orienté vers un serveur syslog ou fichiers.

Syslog

Les journaux doivent être formatés pour contenir dans l'ordre les informations suivantes : la date à laquelle a été émis le journal, le nom de l'équipement ayant généré le journal, une information sur le processus qui a déclenché cette émission, le niveau de priorité du journal, un identifiant du processus ayant généré le journal et enfin un corps de message.

La priorité du message est particulièrement importante car elle permet de déterminer la catégorie et la gravité du journal.

	Gravité	**Mot-clé**	**Description**
0	Emergency	emerg (panic)	Système inutilisable.
1	Alert	alert	Une intervention immédiate est nécessaire.
2	Critical	crit	Erreur critique pour le système.
3	Error	err (error)	Erreur de fonctionnement.
4	Warning	warn (warning)	Avertissement (une erreur peut intervenir si aucune action n'est prise).
5	Notice	notice	Événement normal méritant d'être signalé.
6	Informational	info	Pour information.
7	Debugging	debug	Message de mise au point.

Fichiers journaux

Il est courant d'utiliser un fichier unique recevant l'ensemble des journaux relatifs à une application. Cette approche conduit à produire des fichiers volumineux difficilement exploitables.

Pour pallier ce problème, il est préférable de mettre en place différents fichiers de journaux ou chacun répond à une problématique donnée. Outre le fait de simplifier leur exploitation, cette approche permet aussi d'adapter les politiques associées à chaque usage (changement de verbosité, profondeur des traces…).

- Les journaux applicatifs
- Les journaux de diagnostic

- Les journaux d'audit

3.1.2. Exigences

Collecte

- **Contenu et pertinence** : Les journaux doivent tracer les événements utiles au diagnostic et au métier (erreurs, étapes significatives de traitement) et rien d'autre : ni secret, ni donnée personnelle non nécessaire, ni verbosité technique sans valeur. Les niveaux doivent être employés de façon homogène.

- **Traçabilité des actions** : les actions sensibles (authentification, modification de droits, opérations métier critiques) doivent alimenter un journal d'audit dédié et inaltérable.

Export

Configuration : La configuration de journalisation (niveau, format, horodatage, destinations) doit être externalisée et modifiable sans relivraison de l'application, notamment pour activer temporairement le niveau DEBUG en diagnostic.

Ingestion

Tableaux de bord : chaque application doit disposer, avant sa mise en production, d'au moins un tableau de bord de supervision couvrant les quatre signaux d'or (Four Golden Signals), à savoir, la latence, le trafic, le taux d'erreurs, la saturation.

3.2. Analyse antivirus

Toute application incluant du téléversement de fichiers doit mettre en place une capacité d'analyse antivirus via le protocole ICAP.

3.3. Détection des secrets

Un outil de détection de secrets doit être exécuté à chaque intégration, ainsi qu'à chaque livraison. La présence d'un secret (mot de passe, clé d'API, clé secrète, identifiant, certificat, jeton d'authentification…) est interdite.
Tout secret détecté doit être considéré comme compromis : sa révocation et son renouvellement sont dès lors obligatoires. Par ailleurs, le secret doit être supprimé de tout historique, y compris dans l'historique du dépôt de l'application.

Tout secret détecté dans un artefact (image de conteneur, binaire) entraîne le rejet de la livraison. Le secret est dès lors compromis et doit être révoqué et renouvelé avant toute nouvelle livraison.

3.4. Gestion des vulnérabilités

Tout code et binaire produit doit être analysé afin de vérifier l'absence de CVE, via une analyse des dépendances et la recherche de vulnérabilités connues. Un rapport est attendu au moins à chaque livraison.

- **Scan de vulnérabilités de dépendances (SCA) :** Les dépendances tierces de l'application et les images de conteneurs doivent être analysées à chaque construction au regard des bases de vulnérabilités connues (CVE).

- **Analyse statique de sécurité du code (SAST) :** Le code produit doit faire l'objet d'une analyse statique de sécurité destinée à détecter les failles de sécurité (injections, désérialisation non sûre, cryptographie faible, contrôles d'accès manquants).

3.5. Déploiement en production

Sauf demande explicite de la DSAF/DSI, toute livraison en production doit être exempte de tout composant de développement ou debug, comme par exemple jUnit, phpinfo.php, mode debug, swagger…

4. Normes de développement

4.1. Généralités

Tout ce qui peut être automatisé doit l'être, les actions manuelles peuvent conduire à des oublis ou des erreurs et sont à éviter.

Pour tous les besoins d'administration et d'intégration, l'utilisation de scripts bash ou Powersell selon l'environnement, ainsi que d'outils comme Terraform ou Ansible est recommandée.

Les scripts doivent être idempotent autant que possible, variabilisé et conditionnels, faire les vérifications nécessaires et ne pas être en échec sans journaux d'erreur.

Tout script doit être documenté, à la fois sur les différents blocs du code et sur les conditions d'utilisation.

Tout ce qui peut être variabilisé doit l'être pour faciliter l'évolutivité, et les variables doivent être documentées. Il faut éviter au maximum les valeurs en dur dans le code, et s'assurer de ne pas avoir de données sensibles (secrets, mots de passe…) en clair.

Toutes les valeurs spécifiques à un environnement doivent être centralisées et variabilisées dans un fichier.

Une attention particulière doit être portée aux permissions des fichiers contenant des valeurs sensibles.

Tout code produit doit être analysé avec au moins des outils de référence selon le langage d'analyse de code statique afin de vérifier la bonne syntaxe, le respect des bonnes pratiques et la sécurité du code.

4.2. Principe du « Clean Code »

Le Clean Code est un ensemble de principes et de pratiques visant à produire du code fonctionnel, mais surtout lisible, maintenable et testable.

Il faut donc intégrer cette philosophie dès le début d'un nouveau développement, en se concentrant sur les fonctionnalités essentielles et en supprimant toute complexité inutile.

- **Nommage expressif** : noms significatifs et intentionnels des variables, fonctions et classes
- **Responsabilité unique** : une fonction ou une classe doit avoir une seule responsabilité
- **Découplage** : les classes doivent être aussi indépendantes que possible les unes des autres, afin qu'un changement dans l'une ait un impact minimal sur les autres
- **Commentaires minimaux** : le code doit être auto-documenté grâce à un nommage et une structure claire. Un commentaire n'est nécessaire que pour expliquer la raison d'une décision complexe ou inhabituelle, ou pour avertir d'un danger potentiel.
- **Consistance visuelle** : un code bien formaté, avec des conventions homogènes, est plus facile à lire et à comprendre.
- **Gestion des erreurs** : un code propre traite les erreurs de manière prévisible et explicite, sans laisser de comportement inattendu.
- **Tests unitaires** : un code découplé, avec des fonctions courtes et des responsabilités bien définies, est naturellement plus facile à tester unitairement.

Le Clean Code s'appuie sur plusieurs principes fondamentaux à appliquer :

- **Conception et architecture**
  - **les principes SOLID** visant à structurer le code de façon modulaire, extensible et facile à faire évoluer.
- **Bonnes pratiques d'écriture de code**
  - **DRY** (Don't Repeat Yourself) : permet d'éviter la duplication de code afin de limiter les incohérences et de faciliter la maintenance.
  - **KISS** (Keep It Simple, Stupid) : privilégie la solution la plus simple qui répond au besoin, en évitant toute complexité inutile dans la conception.
  - **YAGNI** (You Aren't Gonna Need It) : préconise le développement des fonctionnalités réellement nécessaires, sans anticiper des besoins futurs hypothétiques.
- **Gestion des erreurs**
  - **Fail-Fast** : permet de détecter et de signaler les erreurs le plus tôt possible dans l'exécution, afin d'éviter qu'elles ne se propagent silencieusement et ne deviennent plus difficiles à diagnostiquer.

4.3. Suivi de version

Tout le code produit, que ce soit pour une application ou des scripts d'exploitation, doit se retrouver dans la forge SPM. Le code doit être relu et validé, et un marqueur (tag) doit être utilisé pour suivre les versions. Toute nouvelle version doit être accompagnée de ses notes de version.

4.4. Stratégie et exigences de tests

Des tests doivent être réalisés pour vérifier la sécurité et la qualité du code produit. Les outils d'intégration continue et de déploiement continu sont à utiliser pour éviter les erreurs humaines, en particulier pour les tests et le packaging des applications.

Il faut donc intégrer les tests dès la conception, documenter la stratégie de test et fournir des indicateurs mesurables de couverture de code.

4.4.1. Typologie de tests

Niveau de tests	**Portée**	**Objectif**	**Fréquence d'exécution**
Tests unitaires	Fonction, méthode, classe	Fiabilité du code	A chaque modification de code
Tests d'API	Contrats d'interface	Conformité et interopérabilité	A chaque livraison de service
Tests d'intégration	Ensemble du système	Fonctionnement de bout en bout	A chaque montée de version Avant une mise en production

Tests unitaires

Un test unitaire valide, de manière isolée, le comportement d'une unité de code (fonction, méthode, classe) indépendamment de ses dépendances externes (base de données, services tiers, systèmes de fichiers…) qui doivent être simulées par des mocks.

- Chaque fonctionnalité livrée doit être couverte par des tests unitaires démontrant son bon fonctionnement, y compris sur les cas limites et les cas d'erreur
- Les tests doivent être indépendants les uns des autres, reproductibles et exécutables sans dépendance à un environnement particulier
- Les frameworks de tests implémentés doivent être conformes aux standards de l'environnement technique. Par exemple, JUnit pour Java, Jest pour Javascript, PHPUnit pour PHP…

Tests d'API

Les tests d'API valident le comportement des interfaces exposées par les services. Ils portent sur le respect du contrat d'interface : formats d'échange, codes de retour, gestion des erreurs, authentification et performance.

- Chaque nouveau endpoint exposé doit faire l'objet de tests couvrant les scénarios nominaux, les cas d'erreur (code 4XX, 5XX), les cas limites (valeurs invalides, manquantes, droits insuffisants…)
- Les contrats d'interface doivent être formalisés dans une documentation OpenAPI et testés, afin de détecter notamment toute régression de compatibilité entre les versions.

Voir chapitre API

Tests d'intégration

Les tests d'intégration valident le fonctionnement du système de bout en bout. Ils visent à détecter les anomalies qui n'apparaissent qu'à l'interface et qui ne peuvent pas être révélées par des tests unitaires isolés.

- Les scénarios d'intégration doivent couvrir les flux métier de bout en bout, sans exposer de données réelles ou sensibles
- Toute intégration avec un système tiers ou existant doit faire l'objet d'un plan de test d'intégration spécifique
- Les tests d'intégration doivent être également rejoués à chaque montée de version d'un composant et à chaque mise en production, afin de détecter d'éventuelles régressions

4.5. Web

Un applicatif web :

- Ne fait pas appel à des ressources externes (hors origine) : CSS, JS, IMG,
- Dissocie les contenus CSS, JS et HTML. Les pages HTML ne comportent pas de CSS ni de JS inline.
- Ne comporte pas d'image inline (encodée en base64),
- N'utilise pas de fonctions Javascript favorisant la vulnérabilité XSS (eval, innerHTML),
- Met en œuvre un contrôle de l'intégrité des JS provenant de sources externes (si disponible et nécessaire).

Toute dérogation à ces règles sera présentée dans le CSP (Content Security Policy) de l'applicatif.

4.6. API

4.6.1. Principe de fonctionnement

Une API (Application Programming Interface) est le point d'accès par lequel une application expose ses données et ses services à d'autres applications.
Les spécifications des API un rôle important dans le partage des conventions. Il existe plusieurs approches pour concevoir des API, et de ces architectures découlent plusieurs spécifications pour standardiser le format des données.

Style architectural

REST (**Re**presentational **S**tate **T**ransfer) est le style architectural de référence pour les API du système d'information.

Il définit un ensemble de contraintes pour la création d'API, qui garantissent un système extensible et évolutif.

REST définit 6 contraintes architecturales qui font de tout service web une véritable API RESTful :

- Le modèle client serveur
- L'interface uniforme
    - L'identification de la ressource
    - L'hypermédia comme moteur de l'état des applications (HATEOAS)
- Le mode sans état (stateless)
- La mise en cache
- Modèle de système multicouche
- Le code à la demande

Format des échanges

Le format des corps de requête et de réponse : il détermine ce que les consommateurs analysent réellement. Son homogénéité permet d'assurer l'interopérabilité.

Versionnement

Le versionnement est le mécanisme qui permet à une API d'évoluer sans casser ses consommateurs : les changements majeurs engendrent une nouvelle version vers laquelle les consommateurs migrent progressivement, pendant une période de coexistence.

Documentation

OpenAPI est le standard de description formelle des API REST : un document (YAML ou JSON) qui décrit exhaustivement les ressources, les opérations, les schémas de données, les codes d'erreur et les modes d'authentification.

4.6.2. Critères d'implémentation

Implémentation systématique et obligatoire requise.

4.6.3. Exigences

Architecture

- **REST par défaut** : toute API exposée doit adopter le style REST. Le recours à un autre style (gRPC, GraphQL) est possible par dérogation spécifique.
- **Orientation ressource** : les URI doivent désigner des ressources uniques. La sémantique des verbes HTTP doit être respectée.
- **Codes de statut normalisés** : Les réponses doivent utiliser les codes HTTP standard selon leur sémantique (2xx succès, 4xx erreur imputable au client, 5xx erreur du serveur).
- **Absence d'état de session** : l'API ne doit conserver aucun état de session côté serveur. Le contexte d'appel est porté par la requête (jeton d'authentification généré par le serveur d'authentification).

Format des échanges

- **JSON** :
    - les corps d'échange doivent être au format JSON (RFC 8259), encodés en UTF-8, avec l'en-tête Content-Type: application/json
- **Conventions de nommage** :
    - les noms de propriétés doivent suivre une convention unique, appliquée sans exception au sein d'une même API.
- **Types normalisés** :
    - les dates et heures doivent être au format ISO 8601 en UTC (2026-07-23T14:30:00Z)
    - les montants monétaires doivent être portés par des types exacts (chaîne ou entier en centimes) accompagnés du code devise ISO 4217
    - les identifiants de ressource ne doivent pas porter de signification métier
- **Format d'erreur unique** :
    - toute réponse d'erreur DOIT suivre le format « Problem Details » (RFC 9457) : un objet JSON normalisé portant le type, le libellé, le statut, le détail, les attributs métier.
    - Les messages d'erreur ne doivent pas exposer d'informations techniques internes (traces d'exécution, requêtes SQL, versions de composants).
- **Pagination, tri, filtrage** :
    - Toute collection de ressources doit être paginée.
    - Les champs (paramètres de pagination, syntaxe de tri et de filtrage) doivent suivre le référentiel commun, identique pour toutes les API du SI.

Versionnement

- **Version majeure :**
    - Dans l'URI : La version majeure doit figurer en préfixe d'URI
    - Dans le header : la version majeure peut être indiquée dans l'entête «Accepted-version ». Toutefois, nous préconisons de spécifier la version dans l'URI.
- **Les changements induisant une nouvelle version majeure** :
    - la suppression ou le renommage d'une propriété ou d'une ressource
    - le changement de type ou de format d'une propriété
    - l'ajout d'un paramètre obligatoire, le durcissement d'une validation.

  Important : les consommateurs doivent être **tolérants en lecture** : l'apparition d'une propriété inconnue dans une réponse ne doit pas provoquer d'erreur.

- **Coexistence et fin de vie : t**oute nouvelle version majeure doit coexister avec la précédente pendant une période annoncée aux consommateurs. La dépréciation doit être signalée techniquement et notifiée aux consommateurs identifiés. Le retrait d'une version sans préavis est interdit.

Documentation OpenAPI

- **Spécification obligatoire et versionnée** : Toute API doit être décrite par une spécification OpenAPI (LST en vigueur), versionnée dans le dépôt de sources. Une divergence constatée entre la spécification et le comportement réel de l'API est un défaut au même titre qu'un bug.
- **Complétude** : La spécification doit couvrir l'intégralité de l'interface : toutes les opérations, tous les schémas, tous les codes d'erreur possibles, et les modes d'authentification.

Livraison : La spécification OpenAPI fait partie du contenu minimal de toute livraison comportant une API.

4.7. Javascript

Les règles de développement CERT sont à appliquer.

4.8. Java

Les règles de développement CERT sont à appliquer.

4.9. PHP

Les standards PSR-1 et PSR-2 sont à appliquer.

4.10. Python

Les recommandations officielles de style de la PEP 8 sont à appliquer.

Concept	Définition
Classe	Une classe est une structure (Building Block) élémentaire permettant de définir le comportement d'un objet et d'instancier celui-ci.
Package	Un package se défini comme un ensemble de classes regroupées sous le même espace de nommage.
Design For Extension	Les méthodes non privées, non statique qui peuvent être surchargée doivent être : -soit abstraites -soit finales -soit avoir une implémentation vide
Loose Coupling	Couplage faible